Fontplant Story

GUST TeX Gyre Team⟨⟩

1 Introduction

1.1 What is Fontplant?

A set of programs and data for creating computer fonts.

1.2 How is it arranged on disk?

The main Fontplant folder is divided into several subfolders. Here is a brief overview of their contents:

Folder	Meaning
`shaper`	Metapost definitions of fonts.
`supervisor`	Python scripts that run Fontplant's machinery.
`mapper`	Data files for renamings, OpenType features, bonds.
`jotter`	Various kind of notes gathered by the authors.
`launcher`	Windows batch files and Linux shell scripts to run Fontplant. Also a convenient location for bonds files.
`narrator`	ConTeXt sources of this story.

Table 1.1 Subfolders of the main Fontplant folder

1.3 How to run it?

The general procedure for running Fontplant follows this schema:

```
$> python <path to runner.py> <input folder> <path to bonds file> <output folder>
```

The Python version invoked by this command must be 3.8 or higher. On Windows, it should be the Python interpreter that comes with the Windows version of FontForge (the standard Python interpreter for Windows will not do!).

1.3.1 A Linux shell script for running Fontplant

Fontplant is provided along with a shell script located at `launcher/fontplant.sh`. Below is a glimpse of its contents:

```
FONTPLANT=... the full path of the fontplant folder on your machine
RESULTS=... the full path of the folder you want the results to be generated

# Set the locale for numbers before calling `fontplant`. This ensures that
# FontForge converts floats to strings using a period instead of a comma.
LC_NUMERIC="en_US.UTF-8"
python $FONTPLANT/supervisor/runner.py $FONTPLANT $FONTPLANT/mapper/$1 $RESULTS
```

Tailor the script according to fit your disk structure and ensure it is executable by using the `chmod a+x` command. Afterward, you can invoke it, passing a bonds file name as a single argument.

The distribution of Fontplant includes bonds files for generating Latin Modern and TEX Gyre fonts. Their names are `bonds-LM.txt` and `bonds-TG.txt`, respectively. Thus, here are the commands to generate the fonts, given you are in fonplant's main catalogue:

```
$> launcher/fontplant.sh bonds-LM.txt
$> launcher/fontplant.sh bonds-TG.txt
```

1.3.2 A Windows batch file for running Fontplant

Running Fontplant on Windows is somewhat peculiar. The peculiarity arises from the fact that the Fontforge system has its own Python interpreter on Windows. To utilize Fontforge's functionality, one must use this specific interpreter, not the standard Python interpreter for Windows.

We recommend creating a Windows batch file, named `ffpython.bat`, to offer a convenient shortcut for executing this specific Python interpreter. Below is an example of how it might appear:

```
:: A batch expected to be in the FontForge directory FontForgeBuilds, e.g.,
::  C:\Program Files (x86)\FontForgeBuilds
:: It is invoked by fonplant.bat.
@echo off
echo Configuring the system path to add FontForge...
set FF=%~dp0
set "PYTHONHOME=%FF%"

if not defined FF_PATH_ADDED (
set "PATH=%FF%;%FF%\bin;%PATH:"=%"
set FF_PATH_ADDED=TRUE
)

for /F "tokens=* USEBACKQ" %%f IN (`dir /b "%FF%lib\python*"`) do (
set "PYTHONPATH=%FF%lib\%%f"
)

:: echo Configuration complete. You can now call 'fontforge' from the console.
:: echo You may also use the bundled Python distribution by calling `ffpython`.
:: echo Extra Python modules, if needed, can be installed via `ffpython`.
"%FF%\bin\ffpython.exe" %*
```

Here is an example of how the `ffpython.bat` batch can be called to run Fontplant. The example is presented in several lines here, in practice it should be a single command line.

```
call "C:\Program Files (x86)\FontForgeBuilds\ffpython.bat" "C:\fontplant\supervisor\runner.py"
  "C:\fontplant"
  "C:\fontplant\mapper\bonds-LM1.txt"
  "C:\fontplant-results\2024-01-14-LM1"
```

In this, the starting script `runner.py` of Fontplant is invoked with three arguments: the input folder, the path to a bonds file and the output folder.

The example content shown above is included with Fontplant in the form of a batch file, named `launcher/fontplant.bat`.

1.4 Understanding the runner

The primary objective of the `runner.py` script revolves around handling command-line arguments and kickstarting Fonplant's sophisticated machinery.

1.4.1 The main routine

The following code represents the routine of the `runner.py` script. The code is executed when the script is called from the command line.

runner 61

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('fontplant_dir')
    parser.add_argument('bonds_file')
    parser.add_argument('output_dir')
    run_args = parser.parse_args()
    run(env_lib.Configuration(run_args.fontplant_dir, run_args.bonds_file, run_args.output_dir))
```

The script expects three mandatory positional arguments, specifically, the paths for: the main fontplant's folder, a bonds file, and the output folder. For convenience, these arguments are encapsulated within a `Conf` object. Subsequently, this data object is passed to the `run` function which orchestrates the operations of Fontplant.

1.4.2 The `run` function

runner 8

Here is the key function which puts Fontplant's machinery into operation:

```
def run(conf: env_lib.Configuration):
    conf.assert_mapper_and_shaper_dirs_exist()
    conf.assert_bonds_file_exists()
    conf.ensure_output_dir()
    bonds_list = bonds_lib.parse_bonds_file(conf.bonds_file_path)
    hub_names_set = bonds_lib.get_hub_names_set(bonds_list)
    conf.archive_previous_output(hub_names_set)
    map_lines_dict = tex_gen.MapLinesDict()
    for bonds in bonds_list:
        locs = env_lib.Locations(conf, bonds)
        locs.ensure_log_dir()
        font_gen.generate_pfb_and_otf_fonts(bonds, locs)
        tex_gen.generate_tfm_and_enc_files(bonds, locs, map_lines_dict)
    tex_gen.generate_map_files(map_lines_dict)
```

The `conf` data object is supposed to contain the paths of: the main Fontplant's folder, a bonds file, and the output folder [☞ 7.1]. The function initiates by verifying the existence of these data, or actually, at the very least, confirming the presence of paths to them.

The function `run` creates also a directory for its output if it does not already exist.

After these basic checks and actions are completed, the system parses the file containing bonds data. The result of this parsing step is stored in the `bonds_list` variable. It is a list of objects of type `Bonds`.

The previous results in the output directory of Fontplant are archived for future reference using the `archive_previous_output` method from the `conf` instance of the `env_lib.Configuration` class [☞ 7.2]. This method determines the storage location of the archive based on data from the bonds data and the current time. Since multiple bonds in the `bond_list` can reference the same hub directory, it's reasonable to copy the old output only once. To accomplish this, the function receives the set of all hub names from the bonds list.

Subsequently, each `bonds` object in `bonds_list` and the run's `conf` argument serves as the foundation for creating an auxiliary `locs`, being an instance of the `Locations` class. Then, the OTF and PFB font files are created, if required by data from the `bonds` object. Additionally, the process involves the creation of TeX-oriented TFM and ENC files.

In the final step, a set of MAP files referencing ENC data, gathered during the iteration over the `bonds`, is generated.

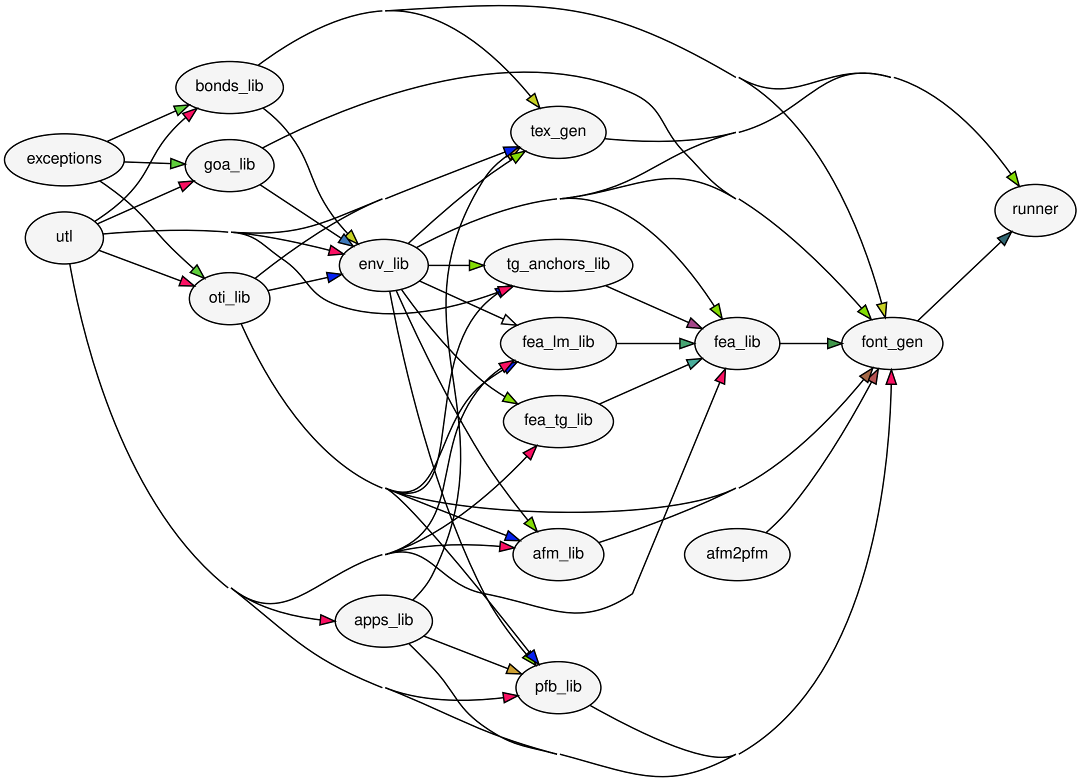
The function `generate_pfb_and_otf_fonts` pertains to a single font, as specified by data from the `bonds` object. Thus, this function remains unaware of other fonts specified in the `bonds_list`. The situation contrasts with the `generate_tfm_and_enc_files` function. In addition to the `bonds` and `locs` objects, this function is supplied with a dictionary containing

MapLines objects. Its responsibility extends to enhancing this dictionary by incorporating encodings specific to the font described in the locs.

1.5 Python scripts of Fontplant

Script	Purpose	☞
runner	The main module of Fontplant. Contains the definition of the function run that sets Fontplant in motion.	1.4
bonds_lib	Parser of bonds files. Defines the Bonds class and several other classes, all making the internal representation of a bonds file.	4
oti_lib	Parser of OTI files. Defines, among others, the OTI and OTI_Glyph classes.	5
goa_lib	Parser of GOADB files.	6
env_lib	Definitions of the Configuration, Locations and Environment classes.	7
font_gen	Supervisor of the font files generating process using the external FontForge system.	8
tg_anchors_lib	Transformer of anchors definitions from an OTI file into parts of a feature file.	TODO
fea_lib	Supervisor of the process of filling a feature file's template with font-specific data.	TODO
fea_lm_lib	Generator of Latin Modern specific parts of a feature file.	TODO
fea_tg_lib	Generator of TeX Gyre specific parts of a feature file.	TODO
tex_gen	Routines for generating TFM, ENC and MAP files, related to Type 1 fonts.	TODO
pfb_lib	Routines for postprocessing PFB files.	TODO
afm_lib	Routines for postprocessing AFM files.	TODO
afm2pfm	A self-standing module for converting an AFM into a PFM file.	TODO
apps_lib	Utilities for running external applications, such as Metapost and those from the t1utils package.	TODO
exceptions	Definitions of a few useful subclasses of the standard Exception class.	TODO
utl	A handful of functions and constants useful in different Fontplant's modules.	3

1.6 Graph of intermodule dependencies



1.7 TODOs

Many aspects of the story you are reading are currently incomplete or missing. These gaps are planned to be filled before Fontplant is made public.

- History of the project.
- Authors (BJ ,PS, PP, MR, RK) and collaborators.
- License.
- Section on other needed and additional helpful software.
- A note on mathematical fonts.
- A note on AFDKO.
- A note on unit tests.
- On the style of imports.
- A note on Python ZEN (Explicit cv. Implicit, Readability counts).
- A section on FontForge and FontLab.
- Stories on Python scripts marked as TODO in section [☞ 1.5].

2 Python code characteristics

In this chapter, we briefly describe the specific notations available in the Python language that are used in Fontplant's code.

2.1 Type annotations

Functions and objects in Fontplant's Python scripts are extensively annotated with types. Although type annotations are optional in Python, the language retains its dynamic typing nature. However, there are tools like the `mypy` library available, which can be utilized to statically check a Python script against types.

Automatic static type-checking can be viewed as automatically generated unit tests. Running a script with invalid types is absolutely not advisable, as it's highly probable that it will yield undesirable results.

Type annotations serve as valuable documentation for code, benefiting not only the reader but also the code's authors themselves. They provide clarity and aid in understanding the codebase.

2.1.1 The `typing` library

While type annotations have become standard in Python since version 3.5, only basic standard types like `int`, `list`, and `dict`, known from earlier versions, can be directly used in annotations without additional steps. For more sophisticated type annotations, the `typing` library must be imported into a script. In Fontplant's code, the following import clause is consistently used:

```
import typing as tp
```

We do not explicitly import specific types using import clauses like this one:

```
from typing import Dict List
```

We prefer the former form, with the `tp` alias, and are not hesitant to utilize dot notation, such as `tp.List[int]` or `tp.Dict[str, tp.List[int]]`. This approach allows us to freely employ any type available in the `typing` library at any point in the code without the need to adjust the import clause when the script evolves and types need to be added or removed.

2.2 Dataclass annotations

The standard Python way of specifying how class instances should be constructed is by using the `__init__(self, ...)` constructor. Example:

```
class C:
   def __init__(self, a: int, s: str):
       self.a = a
       self.s = s
```

Following this declaration, instances of class `C` will have to be created with two arguments, like `C(123, 'abc')` or `C(2 * 3, "a" + "b")`. The `__init__(self, ...)` constructor introduces two attributes of the class: `self.a` and `self.a`, initializing them with the values of its arguments.

For many, it's a boilerplate kind of code. An alternative notation for class declaration is possible, using the (nowadays) standard dataclass annotations. In Fontplant's code, numerous classes are defined using these annotations. For example, here is how the above `C` class could be defined:

```
import dataclasses as dc

@dc.dataclass
class C:
    a: int
    s: str
```

The trick behind the `@dc.dataclass` annotation is that it automatically generates the `__init__` constructor of the class for the programmer, saving their time and improving code readability.

For this story's authors even utilizing `@dc.dataclass` feels like grappling with boilerplate code. Thus, a deliberate choice emerged: in the narrative, Fonplant's Python code is rendered without those annotations, despite their presence within the original sources.

2.3 The type of self

TODO

2.4 Default factories

This section pertains to a topic highly specific to Python. When the Python interpreter reads a script, certain data from the script are stored as invariants for any usage of the functions from the script.

One cannot give a simple syntax for mutable fields. Solution was proposed for that in PEP 671.

TODO

2.5 Special comments for documentation

The careful reader can find a special form of comments in the code. They have the form of lines that start with `#+`, `#:` or `#-`. These comments are processed by software (Lua plus ConTEXt) used for generating the PDF that you are now reading. This means they are used for generating documentation.

Here is an example:

```
#+
#: The bonds file is absolutely vital for Fontplant to work. That's why we
#: define a function to verify its existence on disk.
    def assert_bonds_file_exists(self):
        utl.assert_file_exists(self.bonds_file_path)
#-
```

Not only is it important which characters a special comment line begins with, but the order of these lines is also crucial. Specifically, documentation comment lines are arranged in *chunks* (the name originates from Donald E. Knuth's concept of Literate Programming Style). A chunk starts with a `#+` line, followed by a sequence of `#:` lines, and finally, it concludes with a chunk-closing `#-` line. The Python code being documented can be included between the comment lines of a chunk.

The above single documenting chunk is transformed by ConTEXt, for the purpose of this story, to the following visual form:

env_lib 38

The bonds file is absolutely vital for Fontplant to work. That's why we define a function to verify its existence on disk.

```
def assert_bonds_file_exists(self):
    utl.assert_file_exists(self.bonds_file_path)
```

The vertical text `env_lib 38` on the left margin informs where the chunk is located in Fontplant's source code: it begins at line 38 of the `env_lib.py` Python script.

3 Utilities for use in various contexts

3.1 Useful regular expressions

utl 10 Regular expressions for recognizing integer and fractional numbers:

```
int_re = r'[+-]?\d+'
num_re = r'[+-]?\d+(\.\d*)?'
```

A regexp for recognizing a non-empty sequence of whitespace characters:

```
spc_re = r'\s+'
```

A regexp for recognizing glyph identifiers:

```
gly_re = r'[A-Za-z0-9\._]+'
```

utl 22 The `grp_re` function provided below serves as a utility for constructing a named group regular expression.

Named regular expression groups are a feature native to Python. Such a group begins with a `?P` tag, followed by the name of the group, denoted by `grp` in this context, enclosed within angle brackets `<` and `>`. For more details on this topic please refer to Python's documentation.

The purpose of the group created by the `grp_re` function is to match the regular expression `pattern` that is passed to it as the seccond argument. Additionally, the third argument, `prefix`, also a regular expression, represents a sequence of characters that have to precede the actual `pattern`.

Upon successfully matching a text, the fragment assigned the name `grp` can be accessed in Python using a `match[grp]` expression.

```
def grp_re(grp: str, pattern: str, prefix: str) -> str:
    return prefix + r'(?P<' + grp + r'>' + pattern + r')'
```

3.2 Veryfying the existence of files and directories

utl 145 Fontplant prefers checking if files or directiories vital for its execution exist. Python's `assert` statement serves this purpose. Like in many commonly known languages, the statement evaluates a condition and halts program execution if the result of evaluation is false. Yes, it might seem abrupt to halt Fontplant if a condition isn't met. However, we do it when it's senseless to proceed any further.

```
def assert_file_exists(file_path: str):
    assert os.path.exists(file_path), 'File ' + file_path + ' does not exist'

def assert_dir_exists(dir_path: str):
    assert os.path.exists(dir_path), 'Folder ' + dir_path + ' does not exist'
```

3.3 Creating files and directories

utl 88

The following function verifies if a directory at a given path exists on disk and creates it if it doesn't. It utilizes the `os.makedirs` function. According to the documentation of `os.makedirs`, setting `exist_ok=True` ensures no exception is generated if the directory already exists on disk.

```
def ensure_dir(dir_path: str):
    os.makedirs(dir_path, exist_ok=True)
```

Yet another function not only ensures that a directory exists but also changes the current directory to it. This function is useful when invoking Metapost, as Metapost writes its results to the current directory only. If we want the results to be placed in a directory of our choice, we need to make that directory the current directory.

```
def ensure_and_enter_dir(dir_path: str):
    ensure_dir(dir_path)
    os.chdir(dir_path)
```

Sometimes we know the path for a file to be created, but before calling the OS to create the file, we want to ensure its directory exists.

```
def ensure_dir_for_file(file_path: str):
    ensure_dir(os.path.dirname(file_path))
```

4 The Bonds File

4.1 The concept of a bonds file

A single run of Fontplant is driven by data gathered in a text file which we refer to as a *Bonds File*. This file contains information about the fonts to be generated and the locations of various font components.

Each bonds file consists of one or more sections resembling the following:

```
FNT:qagb HUB:tex_gyre  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
```

Such a section begins with a `FNT:` line, which initially specifies a reference name for a font. The name is the word right after the `FNT:` keyword. This name distinguishes the font from those listed in other sections of the bonds file.

The `FNT:` line is mandatory in each section, while all subsequent lines are optional. These subsequent lines determine the font type — whether OpenType or Type 1 — that Fontplant is to generate, as well as any additional files such as encodings or mappings.

If your interest lies solely in an OpenType font, create a section containing only two lines: `FNT:` and `OTF:`. For a Type 1 font, include a `PFB:` line in your bonds section. In the latter case, it's probable that you will also require Fontplant to generate TEX-related files for the necessary encodings (`MPE:` lines), along with a MAP file listing the generated encodings (`MAP:` line).

During the parsing of a bonds file, Fontplant ignores empty lines and comments. Comments are identified by starting with the `#` character and extending to the end of a line. In the sample section shown above, a comment causes the encoding `e-qx` not to be generated.

A non-empty sequence of whitespace characters means the same for Fontplant as a single space. This means that the many spaces between the `OTF:` and `FEA:` keywords above can be reduced to a single one.

A section must begin with a `FNT:` line. The order of the remaining lines within the section is arbitrary.

4.1.1 Bonds files that come with Fontplant

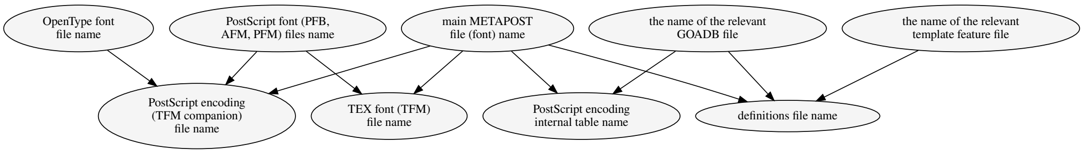
In Fontplant's distribution, users can find the bonds files crafted for generating a diverse range of fonts. These files are located in the `launcher` folder. `launcher/bonds-LM.txt` contains data necessary for generating fonts within the Latin Modern family, while `launcher/bonds-TG.txt` holds data for crafting fonts from the following families of TEX Gyre collection: Adventor, Schola, Pagella, and Termes.

4.2 Why `FNT:` or `FEA:` and not simply `FNT` or `FEA`

The current implementation of Fontplant dictates that the colon characters ':' are included in the keyword names. Therefore, in a bonds file, each no-nempty line comprises a sequence of key-value pairs, rather than a sequence of key-separator-value triples.

4.3 Why the name *Bonds File*?

The term *Bonds File* reflects the fact that its content can be interpreted as representing a graph of relationships between components needed for generating font files.



4.4 The meaning of lines and keywords

Significant lines within a bonds file can be categorized into four distinct groups. The category of each line is identified by its starting keyword. The possible starting keywords are: `FEA:`, `OTF:`, `MPE:` and `MAP:`. The meaning of these keywords as well as of the keywords that follow them in their lines is desribed below.

It is important to note that all keywords following the initial one are mandatory within the line. Additionally, a value associated with any keyword must be a sequence of non-whitespace characters. Whitespace characters before and after the value are ignored.

4.4.1 The `FNT:` line

Keyword	Example	Value
`FNT:`	`qagb`	Font Identifier. Utilized by Fontplant to distinguish a font within the bonds file. Moreover, crucially, the name of a Metapost file containing a script defining the glyph shapes (outlines) of the font.
`HUB:`	`tex_gyre`	The folder name where Metapost scripts of a collection reside, including one named in the `FNT:` line. Fontplant expects Metapost scripts to reside in the `shaper` directory. Thus, the hub folder should be a subdirectory of `shaper`. Additionally, it refers to the directory Fontplant creates and fills with its results. This directory is established as a subdirectory of the location specified for the results in Fontplant's command line invocation.
`GOA:`	`goadb.txt`	The name of a GOADB file. Fontplant expects the file to reside in the `mapper` directory.
`HDR:`	`TG_headers.dat`	The name of a file with texts Fontplant places on top of various font-related files it generates. Fontplant expects the file to reside in the `mapper` directory.

4.4.2 The `OTF:` line

Keyword	Example	Value
`OTF:`	`texgyreadventor-bold`	The name of an OpenType font file which Fontplant is to produce. The name may or may not include the `.otf` extension. If the extension is provided, Fontplant stores the name in its internal structures without it. It does so because FontForge automatically appends the ex-

		tension, which could lead to the generated font file having the `.otf.otf` extension. Each section of a bonds file may contain at most one `OTF:` line.
`FEA:`	`TG_fea.dat`	The name of a template file with the definitions of OpenType features of the file.

4.4.3 The `PFB:` line

Keyword	Example	Value
`PFB:`	`qagb`	The name of a Type 1 font file Fontplant is to produce. Three Type 1 font-related files are generated using the name specified in the `PFB:` line: the font file itself in a `.pfb` format, Adobe font metrics in a `.afm` file, and Windows font metrics in a `.pfm` file. Each section of a bonds file may contain at most one `PFB:` line.

4.4.4 The `MPE:` line

Keyword	Example	Value
`MPE:`	`e-cs`	The symbolic name of an encoding. It also serves as the name of a Metapost script containing the source definition of the encoding. Here, it is provided without an extension; however, Fontplant appends `.mp` as the extension to construct the name of the Metapost script. Fontplant anticipates that the Metapost scripts f or encodings will reside in the `shaper/fontbase` directory. Each section of a bonds file may contain multiple `MPE:` lines. Although the encodings specified in the `MPE:` lines and mappings in a `MAP:` line are closely tied to Type 1 fonts, the inclusion of a `PFB:` line within the section is not obligatory. This is due to the fact that when Fontplant processes a `MPE:` or `MAP:` line, it depends on the symbolic font name and the hub name provided in the `FNT:` line of the section.
`TFM:`	`cs-qagb`	The name of a TFM (TEXFont Metrics) file generated by Metapost for the font.
`PSE:`	`q-cs`	The name designated as the Postscript identifier for the encoding, placed by Fontplant in the MAP file if generated.
`PSI:`	`encqcs`	The internal name of the encoding to be positioned at the top of the encoding file generated by MetaPost. For reference, it also goes to the MAP file.

4.4.5 `MAP:` line

Keyword	Example	Value
MAP:	qag	The name of a MAP file. Fontplant is responsible for creating and populating it with the encodings specified in MPE: lines. There can be only one MAP: line in any section of the bonds file. If the same name is referenced in the MAP: lines of multiple sections of the bonds file, Fontplant generates a single file with the name, containing a concatenation of encodings-related texts from those sections.

4.5 Transforming a bonds file into internal data structures

The complete code responsible for parsing a bonds file and transforming it into internal data structures is encapsulated within the `bonds_lib` module.

4.5.1 Internal representation of bonds

bonds_lib 52

The primary function of this module is to parse the textual data from a bonds file. Through its operations, the lines of the file are converted into the following data structure for storage and further processing.

```
class Bonds:
    fnt: FntBonds
    pfb: tp.Optional[PfbBonds] = None
    otf: tp.Optional[OtfBonds] = None
    map_bonds: tp.Optional[MapBonds] = None
    enc_bonds_list: EncBondsList = dc.field(default_factory=list)
```

This definition pertains to other data structures, such as `FntBonds` and `PfbBonds`. This module encompasses definitions for each of these structures.

It's crucial to note throughout the rest of Fontplant's codebase that an instance of the `Bonds` class stores data associated with only one font. This is what a single section of a bonds file represents.

You might be curious about the unfamiliar phrase `dc.field(default_factory=list)` serving as the default value for the enc_bonds_list field. It creates an empty list. In Python version 3.8, you cannot straightforwardly utilize the conventional constructor notation `EncBondsList()`. This limitation seems to stem from a current weakness in Python's typing mechanisms.

Please note that the `pfb`, `otf`, and `map` fields are declared as optional, unlike the `fnt` field, which is mandatory. This distinction mirrors the requirement that only a `FNT:` line must exist in any section of a bonds file. Although the `enc_bonds_list` is not explicitly typed as optional, its default value is an empty list, which signifies that a bonds file section may not necessarily contain any encodings.

Information for a group of bonds will be kept as a list. For ease of use, a type synonym is established for this list:

```
BondsList = tp.List[Bonds]
```

bonds_lib 11

A series of class definitions follows, each corresponding to a type of line found in a bonds file. All fields within the classes are of type string, and there is no need to convert them to any other type. Furthermore, all fields are mandatory, as for each type of line in a bonds file, it is required that all its fields be present.

```
class FntBonds:
    id: str   # Basic (short for convenience) font identifier
    hub: str  # Subdirectory of `shaper` where fonts of the same family reside
    goa: str  # goadb file name within the `mapper` subdirectory
    hdr: str  # headers file name within the `mapper` subdirectory

class PfbBonds:
    name: str # Name of PFB font file (sans the .pfb extension)

class OtfBonds:
    name: str # Name of OTF font file (sans the .otf extension)
    fea: str  # Name of FEA template's file

class MapBonds:
    name: str # Name of MAP file

class EncBonds:
    mpe: str  # Name of source Metapost script
    tfm: str  # Name of TFM file
    pse: str  # Name of ENC file
    psi: str  # Internal Postsript encoding name for use within encoding file

EncBondsList = tp.List[EncBonds]
```

4.5.2 Parsing a bonds file

bonds_lib 250

Below is the main function for parsing a bonds file. As evident, it reads the entire contents of the file into a string and delegates the task of parsing to the `parse_bonds_file_text` function.

```
def parse_bonds_file(bonds_file_path: str) -> BondsList:
    utl.assert_file_exists(bonds_file_path)
    with open(bonds_file_path, 'r') as bonds_file:
        txt = bonds_file.read()
    return parse_bonds_file_text(txt)
```

bonds_lib 229

Parsing the contents of a bonds file commences with splitting it into lines. The meaningful portion of these lines is evaluated by removing comments and whitespace. This task is executed by the `get_meaningful_lines` function from the `utl` library. It not only eliminates whitespace and comments but also assigns line numbers and excludes empty lines, specifically those rendered empty by the cleaning process.

The sequence of meaningful lines is forwarded to another function, `parse_bonds_file_lines`. This function identifies the grammatical structure within the lines and transforms them into a bonds list. In the event of any errors in the grammatical structure, an error message is printed, and Fontplant is halted.

```
def parse_bonds_file_text(bonds_file_text: str) -> BondsList:
    try:
        meaningful_lines = utl.get_meaningful_lines(bonds_file_text)
        return list(parse_bonds_file_lines(meaningful_lines))
    except exc.BondsError as err:
        print(err.exc_message)
        exit()
```

4.5.3 Line beginnings matter so much…

bonds_lib 172

The ultimate task of recognizing the type and internal structure of bonds file lines is performed by the `parse_bonds_lines` function below. It meticulously inspects each line, focusing on its beginning. It anticipates the line to commence with one of the keywords designated as line starters. Further actions hinge upon the keyword from which the line begins.

```
def parse_bonds_file_lines(num_lines: tp.Iterable[utl.NumLine]) -> tp.Iterable[Bonds]:
    bonds = None
    for num_line in num_lines:
        (line_no, line) = num_line
        if line.startswith("FNT:"):
            # A new bonds section, for a single font, begins.
            # Emit the single font bonds that just have been finished, if any:
            if bonds is not None:
                yield bonds
            # Create a new single font bonds object. It will be filled with
            # the results of parsing the subsequent lines.
            fnt = parse_fnt_line(num_line)
            bonds = Bonds(fnt)
        elif bonds is None:
            raise exc.BondsError(line_no, "A line starting with FNT: expected")
        elif line.startswith("PFB:"):
            if bonds.pfb is not None:
                raise exc.BondsError(line_no, "PFB bonds repeated for a font")
            bonds.pfb = parse_pfb_line(num_line)
        elif line.startswith("OTF:"):
            if bonds.otf is not None:
                raise exc.BondsError(line_no, "OTF bonds repeated for a font")
            bonds.otf = parse_otf_line(num_line)
        elif line.startswith("MAP:"):
            if bonds.map_bonds is not None:
                raise exc.BondsError(line_no, "MAP bonds repeated for a font")
            bonds.map_bonds = parse_map_line(num_line)
        elif line.startswith("MPE:"):
            enc_bonds = parse_mpe_line(num_line)
            bonds.enc_bonds_list.append(enc_bonds)
        else:
            raise exc.BondsError(line_no,
                f"Line should start with one of: {['FNT:', 'PFB:', 'OTF:', 'MAP:', 'MPE:']}")
    # Don't forget to emit the possibly pending last bonds section:
    if bonds is not None:
        yield bonds
```

The function returns an iterator of objects of type `Bonds`. As a reminder, each of these objects represents a bonds file's section associated with a single font. Why an iterator instead of a list or a dictionary? Certainly, any of these alternatives would do, so it's simply a matter of *licencia poetica*.

One of the tasks performed by the function is the verification of certain conditions crucial for subsequent Fontplant steps. For instance, it verifies if a section starts with a `FNT:` line and ensures that the occurrences of `OTF:`, `PFB:`, and `MAP:` are limited to at most once in the section. The disturbance of any of these conditions results in throwing an exception

Once the type of the line has been determined, a dedicated parser for this type is invoked, such as `parse_fnt_line` or `parse_pfb_line`.

4.5.4 Recognizing key-value pairs within a line

bonds_lib 95

The function below parses a line and constructs a dictionary of key-value pairs found within it. It scans the line to identify keys from the `line_keys` list. The value for each key is the text located between that key and the subsequent one (or until the end of the line if there is no next key).

The keys within the line must match the order specified in the `line_keys` list. If they do not, a BondsError exception is raised. It is assumed (though not verified) that the keys in `line_keys` are unique.

```
def parse_line_into_key_vals(line_keys: tp.List[str], num_line: utl.NumLine) -> tp.Dict[str, str]:
    result = {}
    (line_no, line) = num_line
    rest = line
    for key in reversed(line_keys):
        # The code for finding key-value pairs occurs simpler
        # when `keys` are searched for in reversed order.
        (before, sep, after) = rest.partition(key)
        if sep == "":
            # `partition` returns empty `sep` if the separator being searched for is not found
            raise exc.BondsError(line_no, key + " expected and not found at " + before)
        result[key] = after.strip()
        rest = before.strip()
    return result
```

Within the function body, the resulting dictionary is stored in a local variable named result. It is initialized at the outset of the function and progressively populated by the `result[key] =` commands within the loop.

The loop iterates over the elements of the `line_keys` list. Before the loop commences, the source line is duplicated into a local variable named `rest`. Throughout the loop, `rest` holds the portion of the entire line that still requires parsing.

The parsing process relies on a standard Python function called partition. This function seeks a designated word (`key` in this context) within a string and yields a triple: the text preceding the word, a separator, and the text following it. The separator consists of the searched word itself or an empty string. In our case, since we mandate the presence of keys from `line_keys` within the line, the separator cannot be empty. Otherwise, a `BondsError` exception is triggered.

It's worth noting that each value inserted into the result dictionary, by the `result[key] =` command, undergoes stripping of whitespace characters.

bonds_lib 140

The parse_line_into_key_vals function above serves as a utility intended for parsing various types of lines within the bonds file. Below, you'll find a series of corresponding parsers.

```
def parse_fnt_line(num_line: utl.NumLine) -> FntBonds:
    key_vals = parse_line_into_key_vals(["FNT:", "HUB:", "GOA:", "HDR:"], num_line)
    return FntBonds(key_vals["FNT:"], key_vals["HUB:"], key_vals["GOA:"], key_vals["HDR:"])

def parse_pfb_line(num_line: utl.NumLine) -> PfbBonds:
    key_vals = parse_line_into_key_vals(["PFB:"], num_line)
    pfb_name = utl.get_file_name_without_default_ext(key_vals["PFB:"], ".pfb")
    return PfbBonds(pfb_name)

def parse_otf_line(num_line: utl.NumLine) -> OtfBonds:
    key_vals = parse_line_into_key_vals(["OTF:", "FEA:"], num_line)
    otf_name = utl.get_file_name_without_default_ext(key_vals["OTF:"], ".otf")
    return OtfBonds(otf_name, key_vals["FEA:"])

def parse_map_line(num_line: utl.NumLine) -> MapBonds:
    key_vals = parse_line_into_key_vals(["MAP:"], num_line)
    return MapBonds(key_vals["MAP:"])

def parse_mpe_line(num_line: utl.NumLine) -> EncBonds:
    key_vals = parse_line_into_key_vals(["MPE:", "TFM:", "PSE:", "PSI:"], num_line)
    return EncBonds(key_vals["MPE:"], key_vals["TFM:"], key_vals["PSE:"], key_vals["PSI:"])
```

In the parsers above, you'll notice that the values of `PFB:` and `OTF:` are stripped of the `.pfb` and `.otf` extensions, respectively. This is done to simplify subsequent code.

5 The OTI File

OTI files come to life on disk when Fontplant calls upon Metapost to generate font data, such as glyph outlines. The Metapost source scripts included with Fontplant contain commands to record specific font information to a text file. This file is named `<fnt>.oti`, where `<fnt>` represents the font identifier used in the Metapost invocation. In the realm of Fontplant, this identifier is the value associated with the `FNT:` keyword in the bonds file processed by Fontplant. The `<fnt>.oti` is written to the same folder in which Metapost leaves its other results.

To clarify: Fontplant users who solely aim to generate font files from the provided resources need not create an OTI text file on their disk. The responsibility of preparing the OTI file rests with the font creator. It is incumbent upon them to incorporate commands into their Metapost source script to output the necessary information to an OTI file generated by the script.

5.1 Why the name *OTI*?

As B.L.Jackowski, P. Strzelczyk and P.Pianowski write in their BachoTeX 2018 paper:

> The abbreviation OTI stands for Olio Typographic Information file. Olio is a traditional name for a potpourri (it appears, e.g., in Robert Burns's Address to a Haggis – "French ragout or olio").

5.2 A taste of the OTI file contents

OTI files generated by the Metapost scripts provided with Fontplant contain two types of lines: `FNT` lines and `GLY` lines. The former contain general information about a font, while the latter pertain to specific glyphs within the font.

Below is an example illustrating the appearance of `FNT` lines, extracted from a snippet of the `qagb.oti` file generated by Metapost scripts for the Adventor Bold font from the TEX Gyre collection.

```
FNT FONT_NAME TeXGyreAdventor-Bold
FNT FULL_NAME TeXGyreAdventor-Bold
FNT ITALIC_ANGLE 0
FNT FONT_DIMEN1 0
FNT DIMEN_NAME1 (slant)
FNT UNDERLINE_POSITION -93
FNT UNDERLINE_THICKNESS 90
FNT FIXED_PITCH false
```

A line of this type starts with the `FNT` keyword. Subsequently, it includes the name of a font-specific parameter, such as `FULL_NAME` or `UNDERLINE_POSITION`. The remainder of the line comprises the value of the parameter. Fontplant processes and interprets all `FNT` lines, providing FontForge with the information they convey. Certain lines serve to append comments to AFM files linked with Type 1 fonts.

Here is a sample of `GLY` lines from the same file:

```
GLY Aogonek CODE 165
GLY Aogonek EPS 265
GLY Aogonek ANCHOR INBAS BOT_MAIN 632 -383
GLY Aogonek ANCHOR INBAS TOP_MAIN 369 875
GLY Aogonek WD 740 HT 739 DP -250 IC 21 GA 369
GLY Aogonek HSBW 740
GLY Aogonek BBX 7 -250 733 739
GLY Aogonek KPX Oogonek -35
GLY Aogonek KPX Q -35
```

```
GLY Aogonek KPX T -40
```

A line of this sort begins with the GLY keyword, succeeded by the name of a character (glyph), such as Aogonek in this case. The rest of the line contains specific information regarding the character. For instance, BBX represents the bounding box of the glyph, while KPX indicates a correction of the normal kerning distance between Aogonek and another glyph, such as the letter T.

In GLY lines, an OTI file encapsulates information about all characters contained within a font.

The names of the glyphs in GLY lines are created by the font creator, who has the freedom to use arbitrary words. In practical scenarios, these names are typically derived from words commonly used by font designers. However, when a font creator introduces a special kind of glyphs for a specific font, these glyphs should be assigned names, and arbitrary words can be used for this purpose.

Glyph names from an OTI file may not precisely match the names assigned to the glyphs in font files (.otf) or (.pfb) generated by FontForge. Fontplant utilizes the GOADB file to determine the definitive names to be used in font files.

5.3 How is the OTI file's contents processed

If you're curious about how Fontplant reads the contents of a Metapost-generated OTI file and converts it into an internal representation, you should explore the oti_lib module. The module encapsulates the entire process of parsing the OTI file and transforming it into its internal representation in Fontplant – an object of the OTI class [☞ 5.7].

And, you will find a story on the conversion process in this very chapter. This process is arranged in two major steps. The first of them consists of parsing the OTI source file and constructing an abstract syntax tree of the OTI's lines, being an object of the OTI_LINES_AST class [☞ 5.4]. The generated abstract syntax tree becomes the source data for the second step. The task of this step is to transform an object of the OTI_LINES_AST class into an object of the OTI class.

5.3.1 A note on the parsing method – regular expresions suffice

The syntactic structure of the OTI file is pretty straightforward. The file consists of lines, each of which can be adequately described using regular expressions. No advanced methods of parsing are needed, such as YACC-like parser generators or Parsing Expression Grammars (PEG). Fontplant's code snippets quoted in the sections that follow show what particular regular expressions are used to parse specific kinds of lines found in an OTI file.

5.3.2 The OTI_Glyph class

oti_lib 78

A group of classes follows, constituting some of the components of OTI's abstract syntax tree. Each class serves as a container for a specific type of attribute associated with an OTI glyph. Attributes can be single strings or numbers, or they may encompass groups of such entities.

It is noteworthy that we intend to convert the string representation of any number extracted from OTI's GLY line into the appropriate numeric data type. This conversion process is consistently applied to numbers encountered across all types of GLY lines, regardless of whether Fontplant necessitates calculations on these numbers. We believe that this consistent approach enhances the clarity and readability of Fontplant's code.

The glyph name gly from a GLY gly... line is not stored in these attribute classes. It is not lost, however, during the parsing process, as it is stored in a separate data class GlyLine provided below.

```
class Code:
    # GLY i CODE 105
    value: int

class Eps:
    # GLY i EPS 205
    value: int

class Size:
    # GLY Lslash WD 517 HT 739 DP 0 IC 13 GA 159
    wd: float  # width
    ht: float  # height
    dp: float  # depth
    ic: float  # italic correction
    ga: float  # glyph axis (OTI only, not for FontForge)

class BoundingBox:
    # GLY i BBX 54 0 188 767
    x1: float
    y1: float
    x2: float
    y2: float

class Hsbw:
    # GLY i HSBW 240
    value: float

class Math:
    # GLY nabla MATH M
    value: str

class Lig:
    # GLY fi LIG f i
    fst: str  # first component's name (glyph or ligature)
    snd: str  # second component's name (glyph or ligature)

class Kern:
    # GLY Aogonek KPX C -35
    kpx: str    # glyph name of the kern pair's second element
    val: float  # kern value

class Anchor:
    # GLY Acutecomb ANCHOR INACC TOP_MAIN -341 1037
    # GLY Acutecomb ANCHOR TOACC TOP_MAIN -341 852
    # GLY dotlessi ANCHOR INBAS BOT_MAIN 100 -123
    # GLY Acutecomb ANCHOR TOBAS TOP_MAIN -341 852
    kind: str
    name: str
    hor: int
    ver: int
```

The following convenient type synonym encompasses all the possible variants of OTI glyph attributes.

```
GlyAttrib = tp.Union[Code, Eps, Size, BoundingBox, Hsbw, Math, Lig, Kern, Anchor]
```

oti_lib 455

The `OTI_Glyph` class defined below encapsulates all attributes of a specific character named `gly` in the OTI. The `gly` attribute is the only obligatory attribute. Thus, an object can be created with only the glyph name `gly` passed to the creator. Remaining class attributes are declared as either dictionaries or non-dictionaries. The latter default to `None`, while dictionaries default to empty. Fontplant's OTI file parser replaces `None` values and fills dictionaries with appropriate objects evaluated during parsing.

Why aren't dictionaries declared as optional and defaulted to `None` like the other attributes? It's a *licencia poetica* kind of thing, which leads to clearer code in subsequent steps. The strange-looking `dc.field(...)` clauses as the dictionaries' default values are explained in detail in the chapter on Python style.

You may wonder why the class is named `OTI_Glyph` and not simply `Glyph`. It's because there are other glyph objects, the ones used by FontForge. The code of subsequent Fontplant's steps will be clearer if the class name is tagged with its origin.

```
class OTI_Glyph:
    gly: str
    code: tp.Optional[Code] = None
    eps: tp.Optional[Eps] = None
    size: tp.Optional[Size] = None
    bbx: tp.Optional[BoundingBox] = None
    hsbw: tp.Optional[Hsbw] = None
    math: tp.Optional[Math] = None
    ligs: tp.OrderedDict[str, str] = dc.field(default_factory=tp.OrderedDict)
    kerns: tp.OrderedDict[str, float] =  dc.field(default_factory=tp.OrderedDict)
    tobas_anchors: tp.OrderedDict[str, Anchor] = dc.field(default_factory=tp.OrderedDict)
    inacc_anchors: tp.OrderedDict[str, Anchor] = dc.field(default_factory=tp.OrderedDict)
    toacc_anchors: tp.OrderedDict[str, Anchor] = dc.field(default_factory=tp.OrderedDict)
    inbas_anchors: tp.OrderedDict[str, Anchor] = dc.field(default_factory=tp.OrderedDict)

    def contains_outlines(self) -> bool:
        assert self.bbx is not None
        return self.bbx.x1 != 0 or self.bbx.y1 != 0 or self.bbx.x2 != 0 or self.bbx.y2 != 0
```

5.4 The Abstract Syntax Tree of OTI lines

oti_lib 168

The following class sits at the apex of the abstract syntax tree for `GLY` lines.

```
class GlyLine:
    line_no: int          # line number, for error messages
    gly: str              # OTI glyph name
    gly_attr: GlyAttrib   # an attribute of the glyph
```

oti_lib 177

Here is the apex of the abstract syntax tree for `FNT` lines. There are two differences with the `GlyLine` class: a property named `prop` is an attribute instead of a glyph named `gly`, and `value` is just a simple string while `gly_attr` is a complex object.

```
class FntLine:
    line_no: int  # line number, for error messages
    prop: str     # the name of a font header property
    value: str    # the value of the property
```

oti_lib 188

Fontplant utilizes the following group of type synonyms to conveniently name the types of intermediate objects processed during parsing the OTI file.

`GLY` lines are collected within a dictionary `GlyLinesDict`. The keys of this dictionary are glyph names, and for each key `gly`, the associated value is expected to be a list containing all `GLY gly...`. Actually not all but only the non-ligature ones. Those, that is `GLY gly LIG...`, lines are to be stored in a separate list. Also `FNT` lines are expected to be stored in a dedicated list.

```
OTI_Line = tp.Union[FntLine, GlyLine]
```

```
GlyLinesDict = tp.OrderedDict[str, tp.List[GlyLine]]
LigLinesList = tp.List[GlyLine]
AttribLinesList = tp.List[FntLine]
DimenLinesList = tp.List[FntLine]
```

oti_lib 205 Finally, we can specify an Abstract Syntax Tree class to represent the lines from the OTI file:

```
class OTI_LINES_AST:
    attrib_lines_list: AttribLinesList
    dimen_lines_list: DimenLinesList
    gly_lines_dict: GlyLinesDict
    lig_lines_list: LigLinesList
```

The `OTI_LINES_AST` class should not be mistaken for the `OTI` one. It only serves as an intermediate data type in the process of transforming the textual form of the OTI file into a rather complex object of the ultimate class `OTI`.

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

oti_lib 877 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_eps=True` for the function when processing `PFB:` or `OTF:` lines from a bonds file. On the other hand, the value `with_eps=False` is used for generating `ENC` files.

oti_lib 848 The following function undertakes a more intricate task compared to its caller, `parse_oti_file`. It delves into the intricacies of the OTI file internals, transforming them into the resulting object of type `OTI`.

```
def parse_oti_text(oti_text: str, with_epses: bool) -> OTI:
    oti_lines = oti_text_to_oti_lines(oti_text)
    oti_lines_ast = oti_lines_to_oti_lines_ast(oti_lines)
    verify_gly_lines_dict(oti_lines_ast.gly_lines_dict, with_epses)
    oti = OTI(oti_lines_ast)
    oti.verify_encoding()
    oti.verify_ligs_dict()
    oti.fill_lig_dicts_within_glyphs()
    return oti
```

Do one thing at a time. Following this proverb, the function executes its task in several phases.

Initially, it splits the contents of `oti_text` into lines and then transforms them into an Abstract Syntax Tree (AST). Following this, it verifies the syntax tree's contents. If verification succeeds, the function constructs an object of the AST's class `OTI`. Then, additional verifications are performed on the components of the object. Finally, relevant information is generated for the OTI glyphs involved in ligature mechanism.

The line `oti = OTI(oti_lines_ast)` within the above code only appears innocent on the surface, but in reality, quite complex actions lie behind it. It's because the creator utilizes `oti_lines_ast` to build the internal attributes of an `OTI` object: `attribs`, `dimens`, `ligs_dict` and `glyphs_dict`.

oti_lib 416

The following function, invoked by `parse_oti_text`, splits the OTI's file text into lines and returns the list of all meaningful ones, classified appropriately as either `FNT` or `GLY` type.

```
def oti_text_to_oti_lines(oti_text: str) -> tp.List[OTI_Line]:
    try:
        meaningful_lines = utl.get_meaningful_lines(oti_text)
        return list(map(oti_line_text_to_oti_line, meaningful_lines))
    except exc.OTI_Error as err:
        print(err.exc_message)
        exit()
```

oti_lib 385

Regular expressions are adequate for identifying the structure of each meaningful line in the OTI file. Below are several useful regex patterns for discerning whether a line corresponds to the `FNT` or `GLY` type. Additionally, Python's special notation for regex patterns can be employed to ease the extraction of variable parts from the line, including the glyph name, property name, and property value. Thanks to this, for a `FNT` line, we can extract the property name as a string named `prop` and the property value as a string named `value` from a successful match. Similarly, a successful match for a `GLY` line will contain the respective parts under the names `gly`, `prop`, and `value`.

```
prop_re = r'[A-Z_]+[0-9]*'
value_re = r'.+'
fnt_line_re = f'^FNT\\s+(?P<prop>{prop_re})\\s+(?P<value>{value_re})$'
gly_line_re = f'^GLY\\s+(?P<gly>{utl.gly_re})\\s+(?P<prop>{prop_re})\\s+(?P<value>{value_re})$'
```

The function `oti_line_text_to_oti_line` below utilizes these regex patterns. It determines the type of an OTI file line and returns the variable parts of the line encapsulated in their respective data types. Two auxiliary functions, `process_fnt_match` and `process_gly_match`, are used to further investigate the structure of the line and construct the result. Each of them receives the result of matching the respective regexp pattern as an argument.

```
def oti_line_text_to_oti_line(num_line: utl.NumLine) -> OTI_Line:
    (line_no, line) = num_line
    if (fnt_match:=re.match(fnt_line_re, line)) is not None:
        return process_fnt_match(line_no, fnt_match)
    elif (gly_match:=re.match(gly_line_re, line)) is not None:
        return process_gly_match(line_no, gly_match)
    else:
        raise exc.OTI_Error(line_no, f"Wrong syntax of OTI line:\n{line}")
```

oti_lib 430

Once the structure of OTI lines is recognized, an Abstract Syntax Tree can be constructed for them. To enhance convenience, the `GLY` lines category is further subdivided into two subcategories: ligature and non-ligature. Below is the function responsible for this task.

```
def oti_lines_to_oti_lines_ast(fnt_or_gly_lines: tp.Iterable[OTI_Line]) -> OTI_LINES_AST:
    attrib_lines_list = []
    dimen_lines_list = []
    gly_lines_dict = GlyLinesDict()
    lig_lines_list = []
    for line in fnt_or_gly_lines:
        if isinstance(line, FntLine):
            if line.prop.startswith('FONT_DIMEN') or line.prop.startswith('DIMEN_NAME'):
                dimen_lines_list.append(line)
            else:
```

```
                attrib_lines_list.append(line)
        else:
            if line.gly not in gly_lines_dict:
                gly_lines_dict[line.gly] = []  # initialize dictionary's item for `line.gly`
            gly_lines_dict[line.gly].append(line)
            if isinstance(line.gly, Lig):
                lig_lines_list.append(line)
    return OTI_LINES_AST(attrib_lines_list, dimen_lines_list, gly_lines_dict, lig_lines_list)
```

5.5.1 Parsing a single `GLY` line

oti_lib 349 When the parser has learned that the current line starts with a `GLY gly` string it can proceed with parsing the rest of the line. The parser's function `oti_line_text_to_oti_line` utilizes the function provided below for the task. It receives a successful match of a `GLY` line, filled with a glyph name, a property name and a string value of the property. The function's code is pretty simple – all possible variants of the properties are recognized and an appropriate parser for each of them is invoked.

```
def process_gly_match(line_no: int, gly_match: re.Match) -> GlyLine:
    gly = gly_match["gly"]
    prop = gly_match["prop"]
    value = gly_match["value"].strip()
    if prop == "CODE":
        return read_gly_code_line(line_no, gly, value)
    elif prop == "EPS":
        return read_gly_eps_line(line_no, gly, value)
    elif prop == "WD":
        return read_gly_size_line(line_no, gly, value)
    elif prop == "HSBW":
        return read_gly_hsbw_line(line_no, gly, value)
    elif prop == "MATH":
        return read_gly_math_line(line_no, gly, value)
    elif prop == "BBX":
        return read_gly_bbx_line(line_no, gly, value)
    elif prop == "LIG":
        return read_gly_lig_line(line_no, gly, value)
    elif prop == "KPX":
        return read_gly_kern_line(line_no, gly, value)
    elif prop == "ANCHOR":
        return read_gly_anchor_line(line_no, gly, value)
    else:
        raise exc.OTI_Error(line_no, f"Unknown property {prop} of glyph {gly}")
```

We won't delve into the intricate details of the definitions of all `read_gly_*` functions invoked by `process_gly_match`. We are afraid that this would overwhelm the Reader. Instead, we will focus on describing the code for only a select few. Those who are curious can refer to the `oti_lib.py` script for insight into how all the variants are coded. For those impatient we may officially declare here that the code for all variants is similar.

oti_lib 292 When Fontplant calls the function `read_gly_kern_line`, presented below, a few things have already been done with the `GLY...KPX...` line being currently processed. Namely, the glyph name `gly` standing after the `GLY` keyword has been extracted as well as the string `value` standing after `KPX`. These two things are passed to the function as arguments.

The task of the function is to extract the components of a kern from `value` and store them in an object of the `Kern` class. Therefore, it employs a regular expression `kern_re` to extract the second glyph name `kpx` of a kern pair and a value `val`. We anticipate that this operation is successful. If it is not, an exception with an error message is generated.

```
def read_gly_kern_line(line_no: int, gly: str, value: str) -> GlyLine:
    kern_re = f"(?P<kpx>{utl.gly_re})\\s+(?P<val>{utl.num_re})"
    if (match:=re.match(kern_re, value)) is not None:
        kpx = match["kpx"]
        val = float(match["val"])
        return GlyLine(line_no, gly, Kern(kpx, val))
    else:
        raise exc.OTI_Error(line_no, "Wrong syntax of KPX (kern) property")
```

oti_lib 314

The following function recognizes the syntax of OTI's anchor lines. It verifies whether the type of an anchor corresponds to one of OTI's keywords: `INACC`, `TOACC`, `INBAS`, `TOBAS`. An `Anchor` object is then created from the components identified through a successful match of `value` against a regex pattern named `anchor_re`, which defines the expected syntax of `value`. If `value` does not match `anchor_re`, an exception with an error message is generated.

```
def read_gly_anchor_line(line_no: int, gly: str, value: str) -> GlyLine:
    kind_re = '(?P<kind>INACC|TOACC|INBAS|TOBAS)'
    glyph_re = utl.grp_re("name", utl.gly_re, utl.spc_re)
    hor_re = utl.grp_re("hor", utl.int_re, utl.spc_re)
    ver_re = utl.grp_re("ver", utl.int_re, utl.spc_re)
    anchor_re = kind_re + glyph_re + hor_re + ver_re
    match = re.match(anchor_re, value)
    if match:
        kind = match["kind"]
        name = match["name"]
        hor = match["hor"]
        ver = match["ver"]
        return GlyLine(line_no, gly, Anchor(kind, name, int(hor), int(ver)))
    else:
        raise exc.OTI_Error(line_no, "Wrong syntax of ANCHOR property")
```

5.5.2 Parsing a single `FNT` line

oti_lib 338

Once it is determined that the current line begins with a `FNT` keyword, a `FntLine` object can be created using the property name and its corresponding value extracted from the line. Both of these are accessible within the `fnt_match` argument of the subsequent function.

```
def process_fnt_match(line_no: int, fnt_match: re.Match) -> FntLine:
    prop = fnt_match["prop"]
    value = fnt_match["value"].strip()
    return FntLine(line_no, prop, value)
```

5.5.3 Trust but verify

Regardless of our trust in the mechanisms embedded within the Metapost scripts that produce the OTI file, it is prudent to verify certain aspects of its contents. Once the Abstract Syntax Tree of OTI lines is constructed, we can assess the integrity of its contents. The verification carried out by the functions outlined in this section may not be exhaustive, but it covers crucial aspects necessary for subsequent steps.

oti_lib 571

The purpose of the following function is to identify potential errors in the OTI file that cannot be detected by analyzing individual lines. The `gly_lines_dict` argument passed to the function is iterated over glyph by glyph to verify the integrity of glyph-related data.

```
def verify_gly_lines_dict(gly_lines_dict: GlyLinesDict, with_epses: bool):
    for (gly, gly_lines) in gly_lines_dict.items():
        assert len(gly_lines) > 0, f"Empty list of lines for glyph {gly}"
        verify_gly_lines(gly, gly_lines, with_epses)
```


The arguments of the **verify_gly_lines** function provided below pertain to a single glyph, named **gly**. The function examines several fundamental integrity conditions for the glyph. Firstly, none of the essential glyph attributes, namely: **code**, **eps**, **size**, **bbx**, and **hsbw**, should be duplicated. Additionally, if the OTI has been generated with EPS files, then glyph data should encompass all these attributes.

```
def verify_gly_lines(gly: str, gly_lines: tp.List[GlyLine], with_epses: bool):
    code_line = None
    eps_line = None
    size_line = None
    bbx_line = None
    hsbw_line = None
    line_no = 0
    for gly_line in gly_lines:
        line_no = gly_line.line_no
        if isinstance(gly_line.gly_attr, Code):
            if code_line is None:
                code_line = gly_line
            else:
                raise exc.OTI_Error(line_no, "Multiple CODE lines for glyph " + gly)
        if isinstance(gly_line.gly_attr, Eps):
            if eps_line is None:
                eps_line = gly_line
            else:
                raise exc.OTI_Error(line_no, "Multiple EPS lines for glyph " + gly)
        if isinstance(gly_line.gly_attr, Size):
            if size_line is None:
                size_line = gly_line
            else:
                raise exc.OTI_Error(line_no, "Multiple SIZE lines for glyph " + gly)
        if isinstance(gly_line.gly_attr, BoundingBox):
            if bbx_line is None:
                bbx_line = gly_line
            else:
                raise exc.OTI_Error(line_no, "Multiple BBX lines for glyph " + gly)
        if isinstance(gly_line.gly_attr, Hsbw):
            if hsbw_line is None:
                hsbw_line = gly_line
            else:
                raise exc.OTI_Error(line_no, "Multiple HSBW lines for glyph " + gly)
    if code_line is None:
        raise exc.OTI_Error(line_no, f"Missing CODE line for glyph {gly}")
    if with_epses and eps_line is None:
        raise exc.OTI_Error(line_no, f"Missing EPS line for glyph {gly}")
    if with_epses and size_line is None:
        raise exc.OTI_Error(line_no, f"Missing SIZE line for glyph {gly}")
    if with_epses and bbx_line is None:
         raise exc.OTI_Error(line_no, f"Missing BBX line for glyph {gly}")
    if with_epses and hsbw_line is None:
        raise exc.OTI_Error(line_no, f"Missing HSBW line for glyph {gly}")
```

5.6 Data structures for the OTI class components

5.6.1 Font attributes

oti_lib 670 Font attributes consist of pairs comprising a name and its corresponding textual value, typically intended for placement in the header section of a font file. Those present in the OTI file are organized by Fontplant into a directory, where font names serve as keys and their associated values are stored as strings. The dictionary can be constructed from a previously recognized list of `FNT` attribute lines using the provided function below.

```
FontAttribs = tp.OrderedDict[str, str]
def eval_attribs(fnt_lines: AttribLinesList) -> FontAttribs:
    attribs = FontAttribs()
    for line in fnt_lines:
        attribs[line.prop] = line.value
    return attribs
```

5.6.2 Font dimensions

oti_lib 686 An OTI file may include a special type of `FNT` lines designed to represent various font dimensions. These dimensions are valuable in TeX's TFM files and for calculating the Panose classification code for OpenType fonts.

```
class Dimen:
    # FNT FONT_DIMEN14 5.6
    # FNT DIMEN_NAME14 (non-standard: digit width)
    size: float
    name: str
```

Storing the set of font dimensions from the OTI file as a list could be a viable option. However, there is no issue in defining `FontDimens` as a mapping from integers to dimensions. The integers are dimension numbers provided in `FNT` dimension lines, like the 14 in the example above. The dictionary of dimensions can be constructed from a previously recognized list of `FNT` dimension lines using the provided function below.

```
FontDimens = tp.OrderedDict[int, Dimen]
def eval_dimens(fnt_lines: DimenLinesList) -> FontDimens:
    dimens = FontDimens()
    dimen_line_re = r'^(FONT_DIMEN|DIMEN_NAME)(?P<num>\d+)$'
    for line in fnt_lines:
        match = re.match(dimen_line_re, line.prop)
        if match:
            num = match["num"]
            dimen = dimens.get(int(num), Dimen(0.0, ""))
            if line.prop.startswith("FONT_DIMEN"):
                dimen.size = float(line.value)
            else:
                dimen.name = line.value
            dimens[int(num)] = dimen
        else:
            raise exc.OTI_Error(line.line_no, f"Wrong syntax of font dimen line {line.prop}")
    return dimens
```

5.6.3 Font glyphs

oti_lib 582 In the `GlyphsDict` dictionary, each key represents a glyph name, which could originate from either the OTI or FontForge's font, depending on the current processing phase.

Utilizing an ordered dictionary instead of a standard one is pivotal for subsequent processing steps. We prioritize maintaining the sequence of glyphs as they appear in the OTI file when transferring them to FontForge.

The provided method below builds the dictionary of glyphs based on a dictionary of previously recognized glyph lines.

```
GlyphsDict = tp.OrderedDict[str, OTI_Glyph]
def eval_glyphs_dict(gly_lines_dict: GlyLinesDict) -> GlyphsDict:
    glyphs_dict = GlyphsDict()
    for (gly, gly_lines) in gly_lines_dict.items():
        glyphs_dict[gly] = eval_oti_glyph(gly, gly_lines)
    return glyphs_dict
```

oti_lib 601 The following function takes the name `gly` of a glyph and a `gly_lines` parameter. It creates an `OTI_Glyph` object, filled with data present in `gly_lines`. The careful reader will notice that ligature lines are omitted.

```
def eval_oti_glyph(gly: str, gly_lines: tp.List[GlyLine]) -> OTI_Glyph:
    oti_glyph = OTI_Glyph(gly)
    for gly_line in gly_lines:
        attr = gly_line.gly_attr
        if isinstance(attr, Code):
            oti_glyph.code = attr
        if isinstance(attr, Eps):
            oti_glyph.eps = attr
        if isinstance(attr, Size):
            oti_glyph.size = attr
        if isinstance(attr, BoundingBox):
            oti_glyph.bbx = attr
        if isinstance(attr, Hsbw):
            oti_glyph.hsbw = attr
        if isinstance(attr, Math):
            oti_glyph.math = attr
        if isinstance(attr, Kern):
            # Warning: Multiple (gly, kpx) pairs are possible in OTI,
            # BOP's decision is to take only the first of them.
            if attr.kpx not in oti_glyph.kerns:
                oti_glyph.kerns[attr.kpx] = attr.val
        if isinstance(attr, Anchor):
            if attr.kind == "TOBAS":
                oti_glyph.tobas_anchors[attr.name] = attr
            if attr.kind == "INBAS":
                oti_glyph.inbas_anchors[attr.name] = attr
            if attr.kind == "TOACC":
                oti_glyph.toacc_anchors[attr.name] = attr
            if attr.kind == "INACC":
                oti_glyph.inacc_anchors[attr.name] = attr
    return oti_glyph
```

5.6.4 Font ligatures

oti_lib 638 The processing of `LIG` lines by the OTI file parser results in the construction of a dictionary, conveniently named `LigLinesDict`. In this dictionary, each key corresponds to an OTI glyph name representing a ligature (e.g., `fi`, `f_k`).

The dictionary's values store ligature lines from the OTI rather than simpler `Lig` objects. This choice is made to retain the source line number, which aids in generating more informative messages regarding potential errors related to ligatures within the OTI file.

The choice of the `OrderedDict` class for the dictionary is deliberate, because the sequence in which ligatures appear in the OTI could be significant for subsequent processing steps. The task of constructing a `LigLinesDict` dictionary from a list of `LIG` lines is performed by the function provided below. The code is straightforward: each line from the `lig_lines` argument is placed as an item into the resulting dictionary, with the line's `gly` serving as the key."

```
LigLinesDict = tp.OrderedDict[str, GlyLine]
def eval_lig_lines_dict(lig_lines: LigLinesList) -> LigLinesDict:
    lig_lines_dict = LigLinesDict()
    for lig_line in lig_lines:
        lig_gly = lig_line.gly
        if lig_gly in lig_lines_dict:
            exc.OTI_Error(lig_line.line_no, f"Ligature glyph {lig_gly} duplicated")
        else:
            lig_lines_dict[lig_gly] = lig_line
    return lig_lines_dict
```

During the dictionary-building process, the function verifies that none of the ligature glyphs is duplicated in the OTI source file.

oti_lib 766

After placing all the glyphs from the OTI into the font's glyph dictionary, we can verify whether the `fst` and `snd` components of each ligature are indeed glyph names. To accomplish this verification step, we define the following function:

```
def verify_ligs_dict(self):
    for lig_line in self.lig_lines_dict.values():
        lig = lig_line.gly_attr
        assert isinstance(lig, Lig)
        if lig.fst not in self.glyphs_dict:
            exc.OTI_Error(lig_line.line_no,
                          f"Ligature's first component {lig.fst} not a known glyph")
        if lig.snd not in self.glyphs_dict:
            exc.OTI_Error(lig_line.line_no,
                          f"Ligature's second component {lig.snd} not a known glyph")
```

oti_lib 783

The `LigLinesDict` dictionary constructed by the `eval_ligs_dict` function only serves as a convenient intermediate structure. Fontplant subsequently converts its contents into an ultimate form: small dictionaries of ligatures stored within specific glyphs. The following function does the task:

```
def fill_lig_dicts_within_glyphs(self):
    for lig_gly, lig_line in self.lig_lines_dict.items():
        lig = lig_line.gly_attr
        assert isinstance(lig, Lig)
        glyph = self.glyphs_dict[lig.fst]
        glyph.ligs[lig.snd] = lig_gly
```

In practice, the function reverses the order of a `LIG` line from the OTI file to obtain an order suitable for the generated font file. Specifically, for a line in the form: `GLY gly LIG fst snd`, it appends a pair (`snd`, `gly`) to the little ligature dictionary associated with `fst`. This means a shift from saying that the ligature `gly` is built from glyphs `fst` and `snd` to saying that when the character `fst` is followed by `snd`, then this combination should be replaced with the ligature `lig`.

5.7 The `OTI` class

It's time to define the ultimate result of parsing the OTI file, namely the `OTI` class.

oti_lib 721

An object of the OTI class encapsulates information pertaining to a font derived from Metapost scripts, stored in the form of an OTI file. While the OTI file itself contains identical information, possibly accompanied by comments which we omit, the OTI class organizes these data in a convenient manner for subsequent processing steps.

```
class OTI:
    def __init__(self, oti_lines_ast: OTI_LINES_AST):
        self.attribs: FontAttribs = eval_attribs(oti_lines_ast.attrib_lines_list)
        self.dimens: FontDimens = eval_dimens(oti_lines_ast.dimen_lines_list)
        self.lig_lines_dict: LigLinesDict = eval_lig_lines_dict(oti_lines_ast.lig_lines_list)
        self.glyphs_dict: GlyphsDict = eval_glyphs_dict(oti_lines_ast.gly_lines_dict)

        self.notdef_from_mpost: bool = True
        self.ensure_notdef_in_glyphs_dict()
```

By design, the `oti_lines_ast` argument is not stored as a field in the class. However, its components are passed to functions responsible for filling the components of the OTI class: font attributes `attribs`, font dimensions `dimens`, a dictionary of ligatures `ligs_lines_dict`, and – perhaps the most awaited data in any font – a collection of glyphs, here stored in the form of a directory `glyphs_dict`.

For information regarding the mysterious `notdef` things located near the end of the above class header, please refer to Section [☞ ??].

5.8 A glimpse at the matter of encodings

The concept of encodings originates from the pre-Unicode era when character codes had to fit into a single byte. This constraint meant that a character code had to be a number within the range of 0 to 255.

If a font contains more than 256 characters (which is quite common), not all of them can be assigned a unique code within the range of 0 to 255. Traditionally, this issue is resolved by dividing the set of all font characters into groups, each containing no more than 256 characters. Each group is then assigned its particular encoding. A Type 1 font designer should ensure coverage for all characters in the font with appropriate encodings.

The encodings do not necessarily need to be disjoint. For instance, in the case of a Latin script, each encoding will encompass the common Latin letters along with various punctuation characters, such as colons, periods, brackets, and so forth. Differences arise with characters beyond this common set.

A bonds file may contain several ENC lines, each referring to a single encoding. Data from these lines are passed over to Metapost, resulting in the generation of different OTI files. While all these OTI files cover the same set of characters, they contain different encodings for subsets of the set. Characters that are not referred to in the encoding are assigned the code -1 in the OTI file.

In the realm of text processing applications, the pre-Unicode era coincides with the pre-OpenType era of the font world. With the emergence of OpenType fonts, the challenge of one-byte encodings seems a relic of the past. Nevertheless, there remain professionals who favor the older Type 1 font format. FontPlant's focus on encoding issues stems from our commitment to producing Type 1 fonts tailored for professional typesetters.

FontPlant's distribution contains Metapost scripts that define encodings tailored to the Latin Modern and TeX Gyre fonts. The names of these scripts are given in the ENC lines of the `launcher/bonds-LM.txt` and `launcher/bonds-TG.txt` files.

5.8.1 The default encoding

There is a specific encoding for which there is no Metapost script for generating it. It is the default encoding based on the OTI generated when Metapost is called to generate EPS-es.

font_gen 434

Type 1 fonts were designed before the Unicode era, when variuos so-called character encodings were in use, each containing no more than 256 character codes. Although a Type 1 font may contain more than 256 characters, applications (such as TEX) get access to them via encodings. There can be more than one encoding defined for a font. Fontplant contains a special module for generating encodings. The following function creates encoding string based on codes from the OTI and puts it into FontForge's font object `ff_font`.

```
def put_default_enc_from_oti_into_ff_font(self):
    ps_enc_name = "FontSpecific"
    enc_string = self.env.form_enc_string(ps_enc_name, preamble="")
    pfb_enc_file_path = self.locs.form_pfb_enc_file_path()
    utl.ensure_dir_for_file(pfb_enc_file_path)
    with open(pfb_enc_file_path, "w") as enc_file:
        enc_file.write(enc_string)
    # Warning! Only after the `with` instruction is accomplished the written file
    # is properly closed by the operating system. Thus, only now it can be read.
    fontforge.loadEncodingFile(pfb_enc_file_path, ps_enc_name)
    self.ff_font.encoding = ps_enc_name
    # The encoding file is only temporarily needed and it can be removed now.
    # Comment out the following line if you need the file, e.g., for debugging.
    os.remove(pfb_enc_file_path)
```

A few words of explanation. One cannot put an encoding string straight into the `ff_font`. It's because FontForge's API only provides a method `loadEncodingFile` that operates on files. Thus, the evaluated contents of `enc_string` must first be stored in a temporary file before it is loaded into `ff_font`. The name of the temporary file can be arbitrary, actually. Here, it is the string assigned to the local variable `ps_enc_name`.

The `form_enc_string` function the above code makes use of is described in the chapter devoted to the `Env` class [☞ 7.3].

5.8.2 Is the default encoding correct?

oti_lib 743

The following function verifies whether character codes in the OTI file indeed fall within the range from -1 to 255. Also, the function does not allow for duplications of any code above -1.

```
def verify_encoding(self):
    visited_codes = set() # The set of integer character codes visited so far
    for glyph in self.glyphs_dict.values():
        if glyph.code is not None and (code:=glyph.code.value) != -1:
            if not (0 <= code <= 255):
                raise exc.OTI_Error(-1, f"Code {code} not within -1..255 at glyph {glyph.gly}")
            if code in visited_codes:
                raise exc.OTI_Error(-1, f"Duplicate code {code} found at glyph {glyph.gly}")
            visited_codes.add(code)
```

5.9 The `.notdef` character

oti_lib 803

Text viewing applications, including web browsers or PDF viewers may encounter a situation when the text refers to a character not present in the current font. To help the applications cope with the problem it is recommended to equip each font with a special character, named `.notdef`. Font creators usually define its glyph as an empty box or a cross surruonded with a thin frame. It is recommended for this extra character to have the internal number 0 in the font.

Fontplant employs the following approach to the `.notdef` glyph issue: If it locates a glyph with this name in the OTI glyphs directory, it ensures it's placed at the beginning of the directory. This task is accomplished using the `OrderedDict` method `move_to_end`.

```
def ensure_notdef_in_glyphs_dict(self):
    gly = '.notdef'
    if self.glyphs_dict.get(gly) is None:
        self.glyphs_dict[gly] = self.create_artificial_notdef(gly)
        self.notdef_from_mpost = False
    # Make `.notdef` the first (number 0) glyph in `glyphs_dict`.
    # `last=False` means: move to the beginning.
    self.glyphs_dict.move_to_end(gly, last=False)
```

If the font lacks a `.notdef` glyph, Fontplant generates an artificial one as an empty box with predefined parameters.

```
def create_artificial_notdef(self, gly: str) -> OTI_Glyph:
    bbx = BoundingBox(x1=40.0, y1=0.0, x2=440.0, y2=700.0)
    size = Size(wd=480.0, ht=700.0, dp=0.0, ic=0.0, ga=0.0)
    hsbw = Hsbw(480)
    eps = self.eval_eps_for_notdef()
    code = Code(-1)
    return OTI_Glyph(gly, code, eps, size, bbx, hsbw)
```

Like any other character in the OTI glyphs directory, the artificially created `.notdef` character must be assigned a unique EPS code. To prevent duplications, the following function calculates its EPS code as the maximum of existing EPS codes plus 1. The result is `None` when the function is invoked after the second run of Metapost, the one not generating EPSes.

```
def eval_eps_for_notdef(self) -> tp.Optional[Eps]:
    eps = None
    for glyph in self.glyphs_dict.values():
        if glyph.eps is not None:
            gly_eps = glyph.eps.value
            eps = gly_eps if eps is None else max(eps, gly_eps)
    return None if eps is None else Eps(eps + 1)
```

6 The GOADB File

Fontplant's concept of GOADB files is closely tied to OpenType fonts. Allow me to share a story that explains this relationship.

When Microsoft and Adobe Systems announced them as a new font format in the late 1990s, Adobe devised a method to automatically convert fonts from older formats to the new one. They implemented this idea by releasing the Adobe Font Development Kit for OpenType (AFDKO), which comprises a set of tools for constructing OpenType font files from PostScript and TrueType font data.

One of the components of AFDKO is the `MakeOTF` application, written in C. It produces an OpenType font from a source font file and from a few auxiliary text files.

One of the arguments required by `MakeOTF` is a text file named `GlyphOrderAndAliasDB`. Here is how Adobe describes its meaning on **`MakeOTF`'s WEB page**:

> `GlyphOrderAndAliasDB` — this file serves three purposes. One purpose is to establish the glyph ID order in the font. Another is to allow glyphs in the output .otf file to have different names than the glyphs in the input source font data. This permits developers to use friendlier names during development of a font, and use different names in the final OpenType file. The third is to provide Unicode® values for the glyphs. `MakeOTF` will provide Unicode values by default for some glyphs, but not all.

Fontplant adopts this concept and utilizes files that serve similar purposes — referred to as `GOADB` files.

6.1 What Fontplant's GOADB file looks like

Here is a snippet from the `goadb.txt` file that is distributed with Fontplant:

```
space_uni0323          dotbelow               ?          NOT_FOR_OTF PFB_AS_MP
space_uni0326          commaaccent            ?          NOT_FOR_OTF
space_uni032E          brevebelow             ?          NOT_FOR_OTF
space_uni032F          brevebelowinverted     ?          NOT_FOR_OTF
space_uni0330          tildebelow             ?          NOT_FOR_OTF
space_uni0331          macronbelow            ?          NOT_FOR_OTF
space_uni0332          linebelow              ?          NOT_FOR_OTF
spade                  spade                  uni2660
spade                  spadesuitblack         uni2660
spadesuitwhite         spadesuitwhite         uni2664
squared_times.alt      squared_timex          ?
ssuperior              ssuperior              ?
star.alt               born                   ?         PFB_AS_MP
sterling               sterling               uni00a3
```

Coming from Greek and Latin cultures, we are accustomed to reading texts from left to right. However, in this case, it is beneficial to begin reading from the second word in each line. This second word represents the name of a glyph as defined by Metapost scripts for a font. Simply put, it's a user-defined name of a glyph that Fonplant can read from the font's OTI.

The first word in a line denotes the name of a glyph to be used in the font file being created, whether it is an OTF or PFB. With exceptions specified at the end of the line. The `PFB_AS_MP` keyword indicates that in the PFB font file, Fontplant should assign the glyph the name not from the first column but from Metapost, i.e., from the OTI file. The keyword `NOT_FOR_OTF` signifies that the glyph should only be present in the PFB font file and omitted from the OTF file.

The third column of in the GOADB file contains Unicode slots for the glyphs. The question mark `?` in this column means that there is no Unicode slot that can be assigned to a glyph.

The presented functionality of Fontplant's GOADB file somewhat extends/modifies the original Adobe's concept. For example, the order of glyphs in the file is ignored by Fontplant. Also a fourth, optional column has been added (for the keywords `NOT_FOR_OTF` and `PFB_AS_MP`).

6.2 Internal representation

goa_lib 9

An object of the `GOA_ENTRY` class below represents a line in a GOADB file. For ease of use, the line's textual content undergoes transformations. Specifically, the glyph name for a Type 1 font is evaluated by the GOADB file's line parser. The third column is assumed to be parsed into two fields, comprising a textual representation of a Unicode codepoint and its integer counterpart.

```
class GOA_Entry:
    mp_name: str        # Second column
    otf_name: str       # First column
    pfb_name: str       # Evaluated from the first and fourth column
    uni_str: str = '?'  # GOADB's textual representation of unicode slot (or ?)
    uni_int: int = -1   # integer unicode slot converted from uni_str (or -1)
    not_for_otf: bool = False
    pfb_as_mp: bool = False
```

Objects of this class are created by parsing a single line of the file with the `parse_goa_line` function. They're stored in a dictionary, for which we define a type alias.

```
GOA = tp.Dict[str, GOA_Entry]
```

This dictionary serves as the internal representation of the content of a GOADB file. The keys are Metapost, i.e., OTI glyph names (`mp_name` in `GOA_ENTRY`). Since the order of glyphs doesn't matter, representing a GOADB file as a dictionary suffices for practical purposes.

6.3 Parsing a GOADB file

goa_lib 106

The main function, `parse_goa_file`, reads a GOA file specified by its path and delegates the parsing task to another function, `parse_goa_text`, that works on its contents.

```
def parse_goa_file(goadb_path: str) -> GOA:
    with open(goadb_path, 'r') as goadb_file:
        goadb_text = goadb_file.read()
    return parse_goa_text(goadb_text)
```

goa_lib 95

The following function parses a GOA file, converting it into a dictionary of type `GOA`. It starts by splitting the input string into lines and removing comments. Each line is then parsed using `parse_goa_line`, and the result is added to the dictionary under the Metapost glyph name found in the line.

```
def parse_goa_text(txt: str) -> GOA:
    num_lines = utl.get_meaningful_lines(txt)
    goadb_entries = map(parse_goa_line, num_lines)
    return dict([(entry.mp_name, entry) for entry in goadb_entries])
```

6.4 Parsing a single line

goa_lib 35

The function below constructs a regular expression for parsing a single line of a GOADB file.

```
NO_OTF = 'NOT_FOR_OTF'
PFB_MP = 'PFB_AS_MP'

def form_goa_line_re() -> str:
    otf_re =  utl.grp_re("otf", utl.gly_re, r'^')
    mpost_re = utl.grp_re("mp", utl.gly_re, r'\s+')
    four_hex_digits = r'[0-9a-f][0-9a-f][0-9a-f][0-9a-f]'
    unicode_re = utl.grp_re("uni", r'(uni|u1)' + four_hex_digits + r'|\?', r'\s+')
    one_or_both = [NO_OTF, PFB_MP, NO_OTF + r'\s+' + PFB_MP, PFB_MP +r'\s+' + NO_OTF]
    fourth_col = utl.grp_re("fourth_col", r'|'.join(one_or_both), r'\s+')
    optional_fourth_col = r'(' + fourth_col + r')?'
    return otf_re + mpost_re + unicode_re + optional_fourth_col + r'$'
```

goa_lib 65

Here is the parser for a single line of a GOADB file. The parsing method relies on matching a regular expression pattern for the line, defined by the `form_goa_line_re` function.

```
def parse_goa_line(num_line: utl.NumLine) -> GOA_Entry:
    (line_no, line) = num_line
    goa_line_re = form_goa_line_re()
    if (match:=re.match(goa_line_re, line)) is not None:
       mp_name = match.group("mp")
       otf_name = match.group("otf")
       uni_str = match.group("uni")
       fourth_col = match.group("fourth_col")

       not_for_otf = False
       if fourth_col is not None:
            not_for_otf = fourth_col.find(NO_OTF) >= 0

       pfb_as_mp = False
       if fourth_col is not None:
            pfb_as_mp = fourth_col.find(PFB_MP) >= 0

       pfb_name = mp_name if pfb_as_mp else otf_name

       uni_int = uni_str_to_int(uni_str)

       return GOA_Entry(mp_name, otf_name, pfb_name, uni_str, uni_int, not_for_otf, pfb_as_mp)
    else:
       raise exc.GOADB_Error(line_no, "Wrong syntax of GOADB line\n" + line)
```

goa_lib 52

The following auxiliary function converts a `uni_str` string into an integer Unicode codepoint. It anticipates that `uni_str` matches the `unicode_re` pattern as defined in the body of the `form_goa_line_re` function.

```
def uni_str_to_int(uni_str: str) -> int:
    if uni_str.startswith("uni"):
        return int(uni_str[3:], base=16)
    if uni_str.startswith("u1"):
        return int(uni_str[1:], base=16)
    return -1
```

7 Environments

Fontplant makes use of a utility script called env_lib.py, which contains a collection of classes. These classes serve as containers for data components, enabling smooth communication between various modules within Fontplant's framework. For a module that utilizes such a class, it functions as an environment in which the module resides and operates.

As Fontplant evolved during development, we identified three distinct types of useful environments. Below are the Python class headers for these three types.

env_lib 14

The `Configuration` class encapsulates the arguments read from the command line for invoking Fontplant.

```
class Configuration:
    input_dir_path: str
    bonds_file_path: str
    output_dir_path: str
```

env_lib 76

The `Locations` class holds the file and directory paths concerning the current bonds. It provides the answer to a fundamental question for FontPlant: where does it fetch data from and where does it write its results?

```
class Locations:
    conf: Configuration
    bonds: bonds_lib.Bonds
```

env_lib 201

The `Environment` class pertains to a single font. Its constructor requires three arguments: a bonds object for the font, locations derived for the bonds and an already parsed `OTI`. The constructor automatically generates two of its other attributes: the internal representations of a GOADB and headers' template. All these data, encapsulated in the environment, Fontplant passes to its font as well as to TeX and ENC files generating modules.

```
class Environment:
    bonds: bonds_lib.Bonds
    locs: Locations
    oti: oti_lib.OTI

    def __post_init__(self):
        self.goa: goa_lib.GOA = self.read_goa()
        self.hdr: str = self.read_hdr()

    def read_goa(self):
        return goa_lib.parse_goa_file(self.locs.form_goa_file_path())

    def read_hdr(self):
        with open(self.locs.form_hdr_file_path()) as hdr_file:
            return hdr_file.read()
```

7.1 Where are my files?

If you need to learn where Fontplant reads a particular file from or where it writes its results, it's best to refer to the classes `Configuration` and `Locations` in the `env_lib.py` module. Both classes are filled with small functions, many of which are just one-liners, that define or verify the paths of Fontplant's input and output files and directories. To not just speak empty words, here are a few representative examples. Let us start from examples within the `Configuration` class.

env_lib 24

The following functions define the paths of important subdirectories within `input_dir_path`, where Metapost scripts with macros for shaping font glyphs are located.

```
def form_shaper_dir_path(self) -> str:
    return os.path.join(self.input_dir_path, 'shaper')

def form_fontbase_dir_path(self) -> str:
    return os.path.join(self.form_shaper_dir_path(), 'fontbase')
```

env_lib 38

The bonds file is absolutely vital for Fontplant to work. That's why we define a function to verify its existence on disk.

```
def assert_bonds_file_exists(self):
    utl.assert_file_exists(self.bonds_file_path)
```

7.2 The where and how of archivization

env_lib 61

By design, the previous output of Fontplant's operation is archived for future reference. Each subdirectory within the output directory, corresponding to hub names listed in the bonds file, undergoes archivization. The following function manages this task, requiring a list of hubs to be passed to it. The archive is a directory named after the hub's name and the current time.

```
def archive_previous_output(self, hub_names_set: tp.Set[str]):
    for hub_name in hub_names_set:
        hub_dir_path = self.form_output_hub_dir_path(hub_name)
        if os.path.exists(hub_dir_path):
            time_stamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
            archive_dir = hub_dir_path + '_' + time_stamp
            shutil.move(hub_dir_path, archive_dir)
```

env_lib 52

Fontplant generates its results in subdirectories within the main output directory. Each of these subdirectories is named after the corresponding hub listed in the bonds file. The following function calculates the full path of such a subdirectory.

```
def form_output_hub_dir_path(self, hub_name: str) -> str:
    hub_parts = re.split(r'[\\/]', hub_name)
    return os.path.join(self.output_dir_path, *hub_parts)
```

7.3 Generating encoding strings

env_lib 226

```
def form_enc_string(self, ps_enc_name: str, preamble: str) -> str:
    enc_glyph_names = [".notdef" for _ in range(256)]
    for oti_glyph in self.oti.glyphs_dict.values():
        if oti_glyph.code is not None and oti_glyph.code != -1:
            pfb_name = self.goa[oti_glyph.gly].pfb_name
            enc_glyph_names[oti_glyph.code.value] = pfb_name
    enc_glyphs_str = "\n".join(["/" + name for name in enc_glyph_names])
```

```
    starting_bracket = "/" + ps_enc_name + "["
    ending_bracket = "] def"
    return "\n".join([preamble, starting_bracket, enc_glyphs_str, ending_bracket])
```

The function makes use of the contents of the font's OTI and GOADB. The class `Env` is the lowest in the hierarchy of the Fontplant's classes where we have both of these data structures at hand.

8 Preparing and generating a font file

As has already been said our project passes the very task of generating PFB and OTF font files to an external application – FontForge. In this chapter we describe how the project communicates with FontForge to make it generate a font file.

This requires several preparatory steps in which some data structures are prepared and sent to FontForge. As will be shown later in this chapter the results produced by FontForge may require some postprocessing.

8.1 The `Font` class

font_gen 23

The `Font` class is used to prepare a font file using FontForge's capabilities. It stores an `env` attribute and FontForge's font data: `ff_font` and `ff_glyphs`.

`Font` is a generic class. It comprises attributes and methods common to the process of generating both Type 1 as well as OpenType font files. Methods that are specific to these two classes of fonts are gathered in two subclasses of `Font`, respectively, `PFB_Font` and `OTF_Font`.

```
class Font:
    locs: env_lib.Locations
    env: env_lib.Environment

    def __post_init__(self):
        self.ff_font: fontforge.font = fontforge.font()
        self.ff_glyphs = {}  # tp.Dict[str, FontForge.glyph] where the keys are OTI's glyph names
```

The `ff_font` attribute represents a FontForge's font object. It is created once the `Font` class is instantiated – by calling the `fontforge.font` constructor in the body of the `__post_init__` method. In subsequent processing steps, `ff_font` is populated with glyphs and other font attributes. Finally, FontForge is employed to convert the contents of the the font object into a PDF or OTF font file, appropriately to the run.

The role of the `ff_glyphs` dictionary is to gather the glyphs are being put into `ff_font` under their names specified in OTI. The glyph name in `ff_font` differs in many cases from its original name from OTI. The `ff_glyphs` structure is a connection between the two worlds: the glyphs generated by Metapost and their counterpart in the FontForge's font object. The `ff_glyphs` structure is set empty when an instance of `Font` is created and is filled with glyph objects when they are being put into FontForge.

8.2 Setting font header attributes

font_gen 65

Every font has a bunch of global parameters describing its general properties. They are collected in a so-called font header. When FontForge is requested to generate a font file it puts the attributes that have been stored in its object structures in an appropriate place of the generated file. FontForge is good at knowing where to put what in a font file.

The function `set_ff_font_header` serves to set the font header attributes. It puts them in the FontForge's font object.

```
def set_ff_font_header(self):
    self.set_ff_font_header_hard_coded_values()
    self.set_ff_font_header_oti_str_values()
    self.set_ff_font_header_oti_num_values()
    self.set_ff_font_family_attribs()
    self.set_ff_font_ttf_sfnt_data()
    self.set_ff_font_panose()
```

As can be seen, `set_ff_font_header` calls calls a sequence of three auxiliary methods to perform three technically different kind of actions. Their description follows.

The first of the auxiliary methods, `set_ff_font_header_hard_coded_values`, sets a group of attributes that do not depend on the font's OTI. Thus, these attributes are common to all all fonts belonging to the Latin Modern and TEX Gyre families.

```
def set_ff_font_header_hard_coded_values(self):
    self.ff_font.hhea_ascent_add = 0
    self.ff_font.hhea_ascent_add = 0
    self.ff_font.hhea_descent_add = 0
    self.ff_font.os2_winascent_add =1
    self.ff_font.os2_windescent_add = 1
    self.ff_font.os2_typoascent_add = 0
    self.ff_font.os2_typodescent_add = 0
    self.ff_font.os2_winascent = 0
    self.ff_font.os2_windescent = 0
```

You might be curious about the origin of names like `hhea_ascent_add` or `os2_windescent` above. Well, these attribute names belong to the tradition of the font world. They have been designed by font gurus coming from various parts of the world. The names are generally accepted and supported by softwwere offered by the Adobe, Microsoft, Apple and other companies. We should be in line with that tradition within our project if we want our fonts be supported by font-processing software worldwide.

The two remaining functions for setting general font attributes take the values of the attributes from the font's OTI. Namely, the values are taken from the `env.OTI.font_header.attribs` object in which font header attributes are gathered. By their nature, the OTI-dependent attributes are related to a single font.

The reasons we have two functions here are of aesthetic nature. The first of the functions copies the string values of the attributes. The task of copying is passed over to an auxiliary subfunction `ff_str_from_oti`. This one may remove spaces from an attibute's value if requested in its logical `remove_spaces` attribute.

```
def set_ff_font_header_oti_str_values(self):
    attribs = self.env.oti.attribs

    self.ff_font.copyright = attribs["AUTHOR"]
    self.ff_font.fullname = attribs["FULL_NAME"]
    self.ff_font.fontname = attribs["FONT_NAME"]
    self.ff_font.version = attribs["VERSION"]
    self.ff_font.weight = attribs["WEIGHT"]
    # GUST Foundry's decision is to remove spaces from the name of the family
    self.ff_font.familyname = attribs["FAMILY_NAME"].replace(" ", "")
```

The second of OTI-dependent font attributes concerns numerical attributes. It defines and uses a local function `flt_attr` for converting some of font header attributes from the OTI into floating point numbers.

```
def set_ff_font_header_oti_num_values(self):
    attribs = self.env.oti.attribs

    def flt_attr(attrib: str) -> float:
       return float(attribs[attrib])

    self.ff_font.ascent = flt_attr("ADL_ASCENDER")
    self.ff_font.descent = flt_attr("ADL_DESCENDER")
    self.ff_font.design_size = flt_attr("DESIGN_SIZE")
    self.ff_font.hhea_ascent = flt_attr("ASCENDER")
```

```
        self.ff_font.hhea_descent = flt_attr("DESCENDER")
        self.ff_font.hhea_linegap = flt_attr("ADL_LINESKIP")
        self.ff_font.italicangle = flt_attr("ITALIC_ANGLE")
        self.ff_font.os2_typoascent = flt_attr("ASCENDER")
        self.ff_font.os2_typodescent = flt_attr("DESCENDER")
        self.ff_font.os2_typolinegap = flt_attr("ADL_LINESKIP")
        self.ff_font.upos = flt_attr("UNDERLINE_POSITION")
        self.ff_font.uwidth = flt_attr("UNDERLINE_THICKNESS")
```

8.3 Adding glyph objects to FontForge's font

FontForge's API has a method for creating glyph objects in a font. The method is called `createChar`. Here is an excerpt from FontForge's documentation about it:

> Create (and return) a character at the specified unicode codepoint in this font and optionally name it. If you wish to create a glyph with no unicode codepoint, set the first argument to -1 and specify a name.

We have had tremendous problems with creating FontForge glyph objects using the `createChar` method. The issues arise from the fact that FontForge has its own interpretation of the association between character names and their corresponding Unicode codepoints. Even if you call the `createChar` method with a specific (codepoint, name) pair FontForge may change any of them. Invoking `createChar` with the pair `(k, n)` in FontForge may lead to a transformation of the pair into some other `(j, m)`. This is done without giving a notice of warning. The effects of the manipulation become apparent only upon inspecting the generated font.

After conducting numerous experiments, we made the decision to initially create all glyphs in FontForge using provisional names of a kind. FontForge has no knowledge of the meaning behind our provisional names, yet we can still make it work with objects bearing these names. The glyphs in FontForge bear these names until a moment commes in font preparations when the provisional names can be replaced with the right ones (taken from GOADB). It happens that if a glyph name is changed at a late stage then FontForge does not reshuffle unicode code of the glyph. Happy us...

font_gen 382

```
def add_glyphs_to_ff_font(self):
    """Extends the set of glyphs in both `ff_font` and `ff_glyphs`"""
    for oti_glyph in self.env.oti.glyphs_dict.values():
        goa_entry = self.get_glyph_goa_entry(oti_glyph)
        if goa_entry is not None:
            unicode_slot = goa_entry.uni_int
            provisional_glyph_name = oti_glyph.provisional_ff_glyph_name()
            ff_glyph = self.ff_font.createChar(unicode_slot, provisional_glyph_name)
            self.ff_glyphs[oti_glyph.gly] = ff_glyph

def get_glyph_goa_entry(self, oti_glyph: oti_lib.OTI_Glyph) -> tp.Optional[goa_lib.GOA_Entry]:
    return self.env.goa.get(oti_glyph.gly, None)
```

8.4 Assigning outlines to a FontForge's glyph

font_gen 306

FontForge has a method for assigning outlines to a glyph. It is called `importOutlines`. It reads outlines from an EPS file, using the file path provided as an argument. Our method called `set_ff_glyph_outlines` is responsible for preparing data for FontForge's `importOutlines` method and subsequently calls it. It has two arguments: a glyph number `eps_num` from OTI and a `ff_glyph` object.

You might be curious about why the code for `set_ff_glyph_outlines` is so complex. Shouldn't it be as simple as a one-liner that calls `importOutlines` with the path to an EPS file? Things aren't as straightforward due to FontForge's specific requirements for what constitutes an EPS file. It's not enough for the file to just contain correctly prepared EPS data; it must also have the correct file name. Here is the story.

The names of EPS files generated by our Metapost scripts are of the form:

```
<font name>'.['<eps_num>']'
```

In this, <eps_num> is a glyph number that is being stored in the EPS attribute of OTI's representation of a glyph.

If a file's name does not have the '.eps' extension then FontForge does not consider it an EPS file. FontForge interprets a file lacking the '.eps' extension as containing an image rather than EPS data. Therefore, it is necessary to add the '.eps' extension to our EPS file before passing it to FontForge.

And this is what makes the code of the set_ff_glyph_outlines so long. It copies a Metapost-generated file to a temporary one with the .eps extension. This extra file is a temporary one indeed; it is removed once the EPS outlines have been passed to FontForge.

```
def set_ff_glyph_outlines(self, eps_num: int, ff_glyph: fontforge.glyph):
    eps_dir = self.locs.form_eps_dir_path()
    mpost_eps_filename = self.env.bonds.fnt.id + "." + str(eps_num)
    mpost_eps_file_path = os.path.join(eps_dir, mpost_eps_filename)
    eps_filename_with_eps_ext = mpost_eps_filename + ".eps"
    mpost_eps_file_path_with_eps_ext = os.path.join(eps_dir, eps_filename_with_eps_ext)
    if os.path.exists(mpost_eps_file_path_with_eps_ext):
        # If a previous run failed for any reason, there might be a temporary EPS file
        # created during that run. We remove it to prevent potential issues.
        os.remove(mpost_eps_file_path_with_eps_ext)
    shutil.copy(mpost_eps_file_path, mpost_eps_file_path_with_eps_ext)
    ff_glyph.importOutlines(mpost_eps_file_path_with_eps_ext)
    os.remove(mpost_eps_file_path_with_eps_ext)
```

8.5 Putting kerns into FontForge's font

font_gen 397

```
def add_kerns_to_ff_font(self):
    # Begin with some FontForge's magic for its representation of kerns:
    self.ff_font.addLookup(
        "kerns", "gpos_pair", (), (("kern", (("DFLT", ("dflt",)), ("latn", ("dflt",)))),)
    )
    self.ff_font.addLookupSubtable("kerns", "kerns0")

    # Fill FontForge's subtable 'kerns0' with the kerns from OTI:
    for (oti_glyphname, ff_glyph) in self.ff_glyphs.items():
        oti_glyph = self.env.oti.glyphs_dict[oti_glyphname]
        for kpx, val in oti_glyph.kerns.items():
            kern_oti_glyph = self.env.oti.glyphs_dict[kpx]
            # A kern table as built here is based on provisional
            # names from OTI, specifically from `oti_lib.Glyph`.
            ff_kpx_glyph_prov_name = kern_oti_glyph.provisional_ff_glyph_name()
            ff_glyph.addPosSub("kerns0", ff_kpx_glyph_prov_name, int(round(val)))
```

8.6 The PFB_Font class

font_gen 424

The PFB_Font class, extending the generic Font one, contains data and methods specific to preparing Type 1 fonts. That is, it generates a pair of PFB, AFM files for the font. A third of Type 1 font-related files, a PFM file, is generated from the AFM in a separate step.

```
class PFB_Font(Font):
```

8.7 Assigning PFB names to FontForge glyphs

font_gen 465

Here is a method for getting rid of the provisional names that FontForge's characters have been bearing since they were created in FontForge. The method replaces them with names taken from GOADB.

```
def assign_pfb_names_from_goadb_to_ff_glyphs(self):
    for oti_glyph in self.env.oti.glyphs_dict.values():
        ff_glyph = self.ff_glyphs[oti_glyph.gly]
        pfb_glyph_name = self.env.goa[oti_glyph.gly].pfb_name
        ff_glyph.glyphname = pfb_glyph_name
```

9 OpenType features

This is how the AI's ChatGPT application (of April 2024) describes the concept of features in OpenType fonts:

> In OpenType fonts, features refer to sets of typographic enhancements or modifications that alter the appearance or behavior of glyphs in text. These features can include ligatures, kerning, alternate character forms, stylistic sets, and more. They allow for greater flexibility and customization in how text is rendered, enabling designers to achieve specific aesthetic or functional goals. Features are typically defined within the font file and can be activated or deactivated by users or software depending on their needs or preferences.

9.1 Features files' templates

The creators of Fontplant faced a challenge: the features file intended for FontForge is a text document filled with numerous interconnected notations. Each of these notations represents various aspects of font-related data. Where do these data originate? While different fonts possess unique features, there are often many similarities among them. Is it feasible to leverage these similarities and focus solely on the differences?

A possible solution to this challenge is to allow Fontplant to operate on templates of features files. Such a template is a text file comprising both a constant and a variable part. The variable part consists of named slots that can be readily identified programmatically and filled with data tailored to a specific font.

To provide you with an impression of what a features file template might resemble, here's a brief excerpt from the "TG_fea.dat" template tailored for the TeX Gyre fonts.

```
@combb=[# bottom combinings
\brevebelowcomb \brevebelowinvertedcomb  \cedillacomb  \caronbelowcomb \circumflexbelowcomb
\commaaccentcomb \dotbelowcomb \comb_lowline \macronbelowcomb \ogonekcomb \tildebelowcomb];

@letcapbas=[
##TG_LET_CAP_BAS##
];

@letsmabas=[
##TG_LET_SMA_BAS##
];
```

Let's focus on the syntactic structure of this fragment, setting aside its meaning and purpose in the context of OpenType features.

The text is divided into three sections: `@combb`, `@letcapba`, and `@letsmabas`. The first section consists of a list of glyph names, each preceded by a backslash character to distinguish glyph names from regular text. The template is intended to contain glyph names from the OTI [☞ **5.6.3**]. Once the template is appropriately processed and ready for FontForge, Fontplant replaces these OTI glyph names with their corresponding OpenType counterparts specified by GOADB.

The two other sections contain slots, named `##TG_LET_CAP_BAS##` and `##TG_LET_SMA_BAS##`. Typically, the character `'#'` marks the beginning of a comment in features files. Therefore, slots in a template are a special type of comment intended to be identified and replaced with actual text by Fontplant. You can observe in Section [☞ **9.2**] how Fontplant locates slots in a template and substitutes them with appropriate text.

9.2 Dispatcher

fea_lib 172

Here is the apex of the hierarchy of methods for processing a features template file into the actual features file required by FontForge. Its schema is very simple: process the template file using `prepare_fea_string` and write the result to an appropriate location on disk.

```
def generate_fea_file(env: env_lib.Environment) -> str:
    fea_string = prepare_fea_string(env)
    fea_file_path = env.locs.form_otf_features_file_path()
    utl.ensure_dir_for_file(fea_file_path)
    with open(fea_file_path, 'w') as features_file:
        features_file.write(fea_string)
    return fea_file_path
```

fea_lib 158

The following function is the main processor of a features template file. It reads the template file, fills its slots with appropriately evaluated values. Finally it does some cleaning job of the result by using GOADB to replace OTI glyph names with their OpenType correspondents.

```
def prepare_fea_string(env: env_lib.Environment) -> str:
   fea_template_path = env.locs.form_features_template_path()
   with open(fea_template_path, 'r') as fea_template_file:
      fea_template = fea_template_file.read()
   fea_oti_str = fill_slots_in_fea_template(env, fea_template)
   fea_goa_str = replace_oti_glyph_names_with_goadb_ones(env, fea_oti_str)
   return fea_goa_str
```

fea_lib 56

The function `fill_slots_in_fea_template` replaces the known slots in `fea_template` with their calculated values. The values of the slots are calculated lazily, i.e., only after a slot to be filled is found in the template and requires replacement.

```
def fill_slots_in_fea_template(env: env_lib.Environment, fea_template: str) -> str:
    fea_str = fea_template

    slot = '##FontRevision##'
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, prepare_font_revision(env))

    slot = '##TG_MARK_LOOKUPS##'
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, tg_anchors_lib.prepare_mark_lookups_and_feature(env))

    slot = '##TG_MKMK_LOOKUPS##'
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, tg_anchors_lib.prepare_mkmk_lookups_and_feature(env))

    slot = '##TG_GDEF_Simple##'
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, fea_tg_lib.prepare_gdef_simple(env))

    slot = '##TG_GDEF_Ligat##'
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, fea_tg_lib.prepare_gdef_ligat(env, fea_template))

    slot = '##TG_GDEF_Mark##'
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, fea_tg_lib.prepare_gdef_mark(env))

    slot = '##TG_LET_SMA_BAS##'
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, fea_tg_lib.prepare_let_smabas_glyph_names())

    slot = '##TG_LET_CAP_BAS##'
```

```
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, fea_tg_lib.prepare_let_capbas_glyph_names())

    slot = '##TG_LET_CSC_BAS##'
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, fea_tg_lib.prepare_let_cscbas_glyph_names())

    slot = '##TG_LET_SMA_OTH##'
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, fea_tg_lib.prepare_let_smaoth_glyph_names(env))

    slot = '##TG_LET_CAP_OTH##'
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, fea_tg_lib.prepare_let_capoth_glyph_names(env))

    slot = '##TG_LET_CSC_OTH##'
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, fea_tg_lib.prepare_let_cscoth_glyph_names(env))

    slot = '##TG_LET_SMA_OTX##'
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, fea_tg_lib.prepare_let_smaotx())

    slot = '##TG_LET_CSC_OTX##'
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, fea_tg_lib.prepare_let_cscotx())

    slot = '##TG_LET_CAP_OTY##'
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, fea_tg_lib.prepare_let_capoty())

    slot = '##TG_LET_CSC_OTY##'
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, fea_tg_lib.prepare_let_cscoty())

    slot = '##TG_MATHM##'
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, fea_tg_lib.prepare_mathm(env))

    slot = '##TG_MATHT##'
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, fea_tg_lib.prepare_matht(env))

    slot = '##TG_CCMP_DECOMP_SUB##'
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, fea_tg_lib.prepare_dccmp_decomp_sub_lookup(env))

    slot = '##TG_LET_DECOMP##'
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, fea_tg_lib.prepare_let_decomp(env))

    slot = '##SIZE##'
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, fea_lm_lib.prepare_size_group(env))

    slot = '##HHEA##'
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, fea_lm_lib.prepare_hhea_group(env))

    slot = '##OS2##'
    if fea_str.find(slot) >= 0:
       fea_str = fea_str.replace(slot, fea_lm_lib.prepare_os2_group(env))

    return fea_str
```

A critical reader may argue that the code above is suboptimal as the input template text is traversed multiple times with the string searching `find` method. Feel free to replace it with a more efficient version, but keep code clarity in mind.

When devising alternative code solutions, it's worth considering utilizing a data structure like a list of (slot, action) pairs. However, a challenge emerges due to the diverse requirements of functions for different slots in the template, spanning from zero to two arguments.

fea_lib 9

A feature template file may include glyph names preceded by a backslash `\'` character. For example: `\A`, `\aogonek`, `\f_k`. These glyph names are derived from Metapost scripts and are used in an OTI file. Before generating a font file, all these names should be substituted with their corresponding GOADB names.

The function `replace_oti_glyph_names_with_goadb_ones` handles this task. It processes all lines within the template string and applies a dedicated function to each of them. Since the template file might include comments, only the portion of a line before any comment is processed by the function. The comments themselves remain unchanged in the result produced by the function.

```
def replace_oti_glyph_names_with_goadb_ones(env: env_lib.Environment, fea_str: str) -> str:
    new_lines = []
    for fea_line in fea_str.splitlines():
        # Do the job outside comments only
        info_and_comment = fea_line.split('#', maxsplit=1)
        new_line = replace_oti_glyph_names_with_goadb_ones_in_fea_line(
           env, info_and_comment[0])
        if len(info_and_comment) > 1:
            new_line = new_line + '#' + info_and_comment[1]
        new_lines.append(new_line)
    return "\n".join(new_lines)
```

fea_lib 33

A feature template line may contain several OTI glyph names. A regular expression is used to recognize them.

```
def replace_oti_glyph_names_with_goadb_ones_in_fea_line(
      env: env_lib.Environment, fea_line: str) -> str:
    gly_in_fea_re = r'\\' + utl.gly_re
    result = ""
    rest = fea_line
    while match:=re.search(gly_in_fea_re, rest):
       before = rest[:match.start()]
       oti_gly = match.group()[1:]
       after = rest[match.end():]
       goa_gly = env.goa.get(oti_gly, None)
       assert goa_gly is not None,\
           f'No GOADB entry found for glyph name {oti_gly} found in line {fea_line}.'
       result = result + before + goa_gly.otf_name
       rest = after
    return result + rest
```

10 Much ado about ligatures

In the realm of fonts, a ligature refers to a combination of two or more characters into a single glyph. Ligatures are typically used to improve the aesthetics and readability of text by replacing certain combinations of letters that might otherwise collide awkwardly or create visual inconsistencies.

A sample of ligatures, present in a few regular fonts of the TEX Gyre collection, is gathered in the following table. It can be seen how certain letterforms are joined to create single glyphs.

Combination	Adventor	Pagella	Termes	Schola
`fi`	ﬁ	ﬁ	ﬁ	ﬁ
`fl`	ﬂ	ﬂ	ﬂ	ﬂ
`ff`	ﬀ	ﬀ	ﬀ	ﬀ
`ffi`	ﬃ	ﬃ	ﬃ	ﬃ
`fk`	fk	fk	fk	fk
`ae`	æ	æ	æ	æ
`oe`	œ	œ	œ	œ

A font designed for the Latin script typically contains only a few ligatures, if any. Ligatures are primarily found in proportional fonts, where glyphs have varying widths. In such fonts, it sometimes makes sense to create a special glyph to represent a combination of other glyphs that appear rather thin and can look awkward when placed adjacent to each other.

The situation with glyph widths differs in Monospaced fonts, of course. Do ligatures make sense in them? The answer is that the mechanism can be used in text processing as a tricky way of replacing a combination of input characters with a single, different character. For example, in his Computer Modern fonts D.E. Knuth defines a ligature consisting of the `?'` combination to represent the glyph ¿.

Fontplant employs a dual approach to ligatures. On the one hand, the OTI file contains adequate information to place the correct ligatures in Type 1 fonts, specifically within the AFM and TFM files generated by Fontplant for these fonts. On the other hand, the data provided within the OTI file proves inadequate for accurately representing information on ligatures in OpenType's OTF file format. For OpenType fonts, Fontplant necessitates the inclusion of ligature information as part of feature files.

Did we mention FontPlant's dual approach to ligatures? In reality, it's not just dual, it's multifaceted. As you delve further into the code outlined in the subsequent sections, you might begin to wonder: why is there such complexity involved in processing this relatively small number of characters? The reasons lie in the history of the development of font methodology and in the ways of solving font-related problems by GUST Fontplant predecessors: MetaType1 and Algotype.

10.1 Processing ligatures for Type 1 fonts

Fontplant extracts information regarding ligatures for Type 1 fonts from the OTI file. During parsing, it captures the content of the `LIG` lines and stores them in internal structures. These structures are then utilized within a dedicated Fontplant's module to postprocess an AFM file generated by FontForge.

A separate path of processing ligatures by Fontplant is related to TFM files. A TFM file is needed by TEX each time it has to typeset a document using a Type 1 font. Fontplant's way of generating TFM files is by using Metapost. One can invoke Metapost with an option telling it to generate TFM. And Fontplant does so.

Fontplant employs a separate pathway for processing ligatures when it comes to generating TFM files. The typesetting system TeX requires information stored in a TFM file whenever it undertakes the task of typesetting a document using a Type 1 font. Fontplant adopts a methodology for TFM file generation through Metapost. Users can initiate Metapost with an option that prompts it to generate a TFM file, a process that Fontplant executes diligently. The ligatures stored in the OTI file aren't necessary for this process. In fact, any OTI file we process originates from Metapost, which inherently includes the content of the OTI when processing the source script files of a font.

10.1.1 Adding ligatures to AFM files

Fontplant does not append information regarding ligatures to a FontForge's font object while generating a Type 1 font. Rather, it enhances the AFM file generated by FontForge with ligatures. Specifically, if a character starts one or more ligature combinations then its line in the AFM file ends with a sequence of those combenations.

For example, here is how a line for the character `f` in AFM may look like:

```
C 102 ; WX 280 ; N f ; B 10 0 274 739 ; L f ff ; L i fi ; L k f_k ; L l fl ;
```

The ligature combinations are at the end of this line. Each is a substring that begins with the letter `L`, followed by two glyph names.

afm_lib 79

The following function constructs a string of ligature combinations to be placed at the end of an AFM's glyph line. This is where the glyph's little ligature dictionary, created during the processing of the OTI file, is utilized by Fontplant. The function returns an empty string if this dictionary is empty.

```
def form_afm_char_ligs_str(oti_glyph: oti_lib.OTI_Glyph) -> str:
    # Sample ligatures string (for 'f'): L k f_k ; L f ff ; L l fl ; L i fi ;
    ligs_list = [" L " + lig[0] + " " + lig[1] + " ;" for lig in oti_glyph.ligs.items()]
    return "" if len(ligs_list) == 0 else "".join(ligs_list)
```

10.2 Processing ligatures for OpenType fonts