Fuzzing with 💃 Fandango

Tutorial and Reference

José Antonio Zamudio Amaya
Marius Smytzek
Andreas Zeller

Find Fandango at
https://fandango-fuzzer.github.io/

CONTENTS

Part I

About Fandango

CHAPTER

ONE

ABOUT FANDANGO

Given the specification of a program's input language, Fandango quickly generates myriads of valid sample inputs for testing.

The specification language combines a *grammar* with *constraints* written in Python, so it is extremely expressive and flexible. Most notably, you can define your own *testing goals* in Fandango. If you need the inputs to have particular values or distributions, you can express all these right away in Fandango.

Fandango supports multiple modes of operation:

- By default, Fandango operates as a *black-box* fuzzer - that is, it creates inputs from a `.fan` Fandango specification file.
- If you have *sample inputs*, Fandango can *mutate* these to obtain more realistic inputs.

Fandango comes as a portable Python program and can easily be run on a large variety of platforms.

Under the hood, Fandango uses sophisticated *evolutionary algorithms* to produce inputs, it starts with a population of random inputs, and evolves these through mutations and cross-over until they fulfill the given constraints.

Fandango is in active development! Features planned for 2025 include:

- protocol testing
- coverage-guided testing
- code-directed testing
- high diversity inputs

and many more.

1.1 Acknowledgments

CISPA
HELMHOLTZ-ZENTRUM FÜR
INFORMATIONSSICHERHEIT

Fandango is a project of the CISPA Helmholtz Center for Information Security[1] to facilitate highly efficient and highly customizable software testing.

Funded by the European Union

European Research Council
Established by the European Commission

This research was funded by the European Union (ERC "Semantics of Software Systems", S3[2], 101093186). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

[1] https://www.cispa.de/
[2] https://www.cispa.de/s3

CHAPTER TWO

RELEASE NOTES

This document lists major changes across releases.

Fandango 0.8 (April 10, 2025)
First public beta release.

CHAPTER

THREE

FANDANGO FAQ

This document holds frequently asked questions about Fandango.

Why not use Python (or any program) to generate outputs in the first place?
Regular programs either *parse* or *produce* inputs. Fandango specifications (including constraints) allow for both in a single, concise representation. Furthermore, you do not have to deal with implementing an appropriate algorithm to achieve goals such as constraints or input diversity; Fandango does all of this for you.

What's the difference to coverage-guided fuzzing?
A specification-based fuzzer such as Fandango is a *blackbox* fuzzer. It does not require feedback (such as coverage) from the program to be tested, nor does it require sample inputs. On the other hand, the constraints used by Fandango do not preclude coverage guidance. Stay tuned for future extensions.

When will Fandango be ready?
We expect a public beta early April 2025, and a 1.0 release end of June 2025.

CHAPTER

FOUR

CONTRIBUTING TO FANDANGO

Welcome! Fandango is a community project that aims to work for a wide range of developers. If you're trying out Fandango, your experience and what you can contribute are important to the project's success.

4.1 Code of Conduct

Everyone participating in Fandango, and in particular in our issue tracker, pull requests, and chat, is expected to treat other people with respect and more generally to follow the guidelines articulated in the Python Community Code of Conduct[3].

4.2 Getting Started with Development

4.2.1 Step 1: Fork the Fandango repository

Within GitHub, navigate to the Fandango GitHub repository[4] and fork the repository.

4.2.2 Step 2: Clone the Fandango repository and enter into it

```
$ git clone https://github.com/fandango-fuzzer/fandango.git
```

```
$ cd fandango
```

4.2.3 Step 3: Create, then activate a virtual environment

```
$ python3 -m venv venv
```

```
$ source venv/bin/activate
```

For Windows, use

```
$ python -m venv venv
```

[3] https://www.python.org/psf/codeofconduct/
[4] https://github.com/fandango-fuzzer/fandango

```
$ . venv/Scripts/activate
```

For more details, see https://docs.python.org/3/library/venv.html#creating-virtual-environments

4.2.4 Step 4: Install development tools

Install `antlr` and other system-level tools:

```
$ make system-dev-tools
```

Install `jupyter-book` and other Python tools:

```
$ make dev-tools
```

Install `pytest` and required files:

```
$ make install-test
```

4.2.5 Step 5: Install Fandango

Install your local copy of Fandango:

```
$ python -m pip install -e .
```

Reset the shell PATH cache (not necessary on Windows):

```
$ hash -r
```

That's it! You just have installed your personal copy of Fandango. You can invoke it as `fandango` and can alter its code as you like.

4.3 Running Tests

Running the full test suite can take a while, and usually is not necessary when preparing a pull request. Once you file a pull request, the full test suite will run on GitHub. You'll then be able to see any test failures, and make any necessary changes to your pull request.

However, if you wish to do so, you can run the full test suite like this:

```
$ make tests
```

or simply

```
$ pytest
```

4.4 First Time Contributors

If you're looking for things to help with, browse our issue tracker[5]!

You do not need to ask for permission to work on any of these issues. Just fix the issue yourself, *try to add a unit test* (page 10) and *open a pull request* (page **??**).

To get help fixing a specific issue, it's often best to comment on the issue itself. You're much more likely to get help if you provide details about what you've tried and where you've looked (maintainers tend to help those who help themselves).

4.5 Contributing Code

Even more excellent than a good bug report is a fix for a bug, or the implementation of a much-needed new feature. We'd love to have your contributions.

We use the usual GitHub pull-request flow, which may be familiar to you if you've contributed to other projects on GitHub. For the mechanics, see GitHub's own documentation[6].

Anyone interested in Fandango may review your code. One of the Fandango core developers will merge your pull request when they think it's ready.

If your change will be a significant amount of work to write, we highly recommend starting by opening an issue laying out what you want to do. That lets a conversation happen early in case other contributors disagree with what you'd like to do or have ideas that will help you do it.

The best pull requests are focused, clearly describe what they're for and why they're correct, and contain tests for whatever changes they make to the code's behavior. As a bonus these are easiest for someone to review, which helps your pull request get merged quickly! Standard advice about good pull requests for open-source projects applies.

4.5.1 Changing Parser Files

If your contribution involves changing the ANTLR `.g4` parser files, you need to recreate the parser code.

Use

```
$ make parser
```

to recreate the parser code.

4.5.2 Contributing to Documentation

If you want to contribute to this very tutorial and reference, be sure to preview your changes. Use

```
$ make html
```

to create a HTML version of the documentation in `docs/_build/html`.

[5] https://github.com/fandango-fuzzer/fandango/issues
[6] https://help.github.com/articles/using-pull-requests/

4.6 Attributions

This guide was forked from the mypy contribution guide[7], which is licensed under the terms of the MIT license.

[7] https://github.com/python/mypy/blob/master/CONTRIBUTING.md

Part II

Fandango Tutorial

CHAPTER

FIVE

FANDANGO TUTORIAL

Welcome to Fandango! In this tutorial, you will learn how to

- write *Fandango specs*
- use the `fandango` *command-line tool*
- *test programs* with Fandango.

Time is precious, so let's get started!

CHAPTER

SIX

INSTALLING FANDANGO

To install Fandango, you first need to install Python[8]. Then, run the following command:

```
$ pip install fandango-fuzzer
```

Note

In this tutorial, a `$` at the beginning of a command stands for your input prompt. Do not enter it yourself.

You can check if everything works well by running

```
$ fandango --help
```

Entering `fandango --help` should result in an output like this:

```
usage: fandango [-h] [--version] [--verbose | --quiet]
                {fuzz,parse,shell,help,copyright,version} ...

The access point to the Fandango framework

options:
  -h, --help            show this help message and exit
  --version             show version number
  --verbose, -v         increase verbosity. Can be given multiple times (-vv)
  --quiet, -q           decrease verbosity. Can be given multiple times (-qq)

commands:
  {fuzz,parse,shell,help,copyright,version}
                        the command to execute
    fuzz                produce outputs from .fan files and test programs
    parse               parse input file(s) according to .fan spec
    shell               run an interactive shell (default)
    help                show this help and exit
    copyright           show copyright
    version             show version

Use `fandango help` to get a list of commands.
Use `fandango help COMMAND` to learn more about COMMAND.
See https://fandango-fuzzer.github.io/ for more information.
```

If this did not work, try out an alternate option; see *Installing Fandango* (page 131).

[8] https://www.python.org/

CHAPTER

SEVEN

A FIRST FANDANGO SPEC

To create test inputs, Fandango needs a *Fandango spec* – a file that describes how the input should be structured.

A Fandango specification contains three types of elements:

- A *grammar* describing the *syntax* of the inputs to be generated;
- Optionally, *constraints* that specify additional properties; and
- Optionally, *definitions* of functions and constants within these constraints.

Only the first of these (the *grammar*) is actually required. Here is a very simple Fandango grammar that will get us started:

```
<start> ::= <digit>+
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

This grammar defines a *sequence of digits*:

- The first line in our grammar defines `<start>` – this is the input to be generated.
- What is `<start>`? This comes on the right-hand side of the "define" operator (`::=`).
- We see that `<start>` is defined as `<digit>+`, which means a non-empty sequence of `<digit>` symbols.

In Fandango grammars, you can append these operators to a symbol:

- `+` indicates *one or more* repetitions;
- `*` indicates *zero or more* repetitions; and
- `?` indicates that the symbol is *optional*.

- What is a `<digit>`? This is defined in the next line: one of ten alternative strings, separated by `|`, each representing a single digit.

To produce inputs from the grammar, Fandango

- starts with the start symbol (`<start>`)
- repeatedly replaces symbols (in `<...>`) by *one of their definitions* (= one of the alternatives separated by `|`) on the right-hand side;
- until no symbols are left.

So,

- `<start>` first becomes `<digit><digit><digit>...` (any number of digits, but at least one);

- each `<digit>` becomes a digit from zero to nine;
- and in the end, we get a string such as `8347`, `66`, `2`, or others.

Let us try this right away by invoking Fandango (page 21).

CHAPTER

EIGHT

INVOKING FANDANGO

8.1 Running Fandango on our Example Spec

To run Fandango on *our "digits" example* (page 19), create or download a file digits.fan with the "digits" grammar in the current folder.

Fandango specs have a `.fan` extension.

Then, you can run Fandango on it to create inputs. The command we need is called `fandango fuzz`, and it takes two important parameters:

- `-f` followed by the `.fan` file – for us, that is `digits.fan`;
- `-n` followed by the number of inputs we want to have – say, 10.

This is how we invoke Fandango:

```
$ fandango fuzz -f digits.fan -n 10
```

And this is what we get:

```
9396
81
37666
977
8194
13941
505
8
3
842
```

Success! We have created 10 random sequences of digits.

Danger

Be aware that `.fan` files can contain Python code that is *executed when being loaded*. This code can execute arbitrary commands.

Caution

Only load `.fan` files you trust.

8.2 More Commands and Options

Besides `fandango fuzz`, Fandango supports more commands, and besides `-f` and `-n`, Fandango supports way more options. To find out which commands Fandango supports, try

```
$ fandango --help
```

Warning

Fandango commands not detailed in this documentation are *experimental* – do not rely on them.

To find out which option a particular command supports, invoke the command with `--help`. For instance, these are all the options supported by `fandango fuzz`:

```
$ fandango fuzz --help
```

```
usage: fandango fuzz [-h] [-f FAN_FILE] [-c CONSTRAINT] [--no-cache]
                     [--no-stdlib] [-s SEPARATOR] [-I DIR] [-d DIRECTORY]
                     [-x FILENAME_EXTENSION]
                     [--format {string,bits,tree,grammar,value,repr,none}]
                     [--file-mode {text,binary,auto}] [--validate]
                     [-S START_SYMBOL] [--warnings-are-errors]
                     [-N MAX_GENERATIONS] [--population-size POPULATION_SIZE]
                     [--elitism-rate ELITISM_RATE]
                     [--crossover-rate CROSSOVER_RATE]
                     [--mutation-rate MUTATION_RATE]
                     [--random-seed RANDOM_SEED]
                     [--destruction-rate DESTRUCTION_RATE]
                     [--max-repetition-rate MAX_REPETITION_RATE]
                     [--max-repetitions MAX_REPETITIONS]
                     [--max-node-rate MAX_NODE_RATE] [--max-nodes MAX_NODES]
                     [-n NUM_OUTPUTS] [--best-effort] [-i INITIAL_POPULATION]
                     [-o OUTPUT] [--input-method {stdin,filename}]
                     [command] ...

options:
  -h, --help            show this help message and exit
  -f FAN_FILE, --fandango-file FAN_FILE
                        Fandango file (.fan, .py) to be processed. Can be
                        given multiple times. Use '-' for stdin
  -c CONSTRAINT, --constraint CONSTRAINT
                        define an additional constraint CONSTRAINT. Can be
                        given multiple times.
  --no-cache            do not cache parsed Fandango files.
  --no-stdlib           do not use standard library when parsing Fandango
                        files.
  -s SEPARATOR, --separator SEPARATOR
                        output SEPARATOR between individual inputs. (default:
```

(continues on next page)

(continued from previous page)

```
                        newline)
  -I DIR, --include-dir DIR
                        specify a directory DIR to search for included
                        Fandango files
  -d DIRECTORY, --directory DIRECTORY
                        create individual output files in DIRECTORY
  -x FILENAME_EXTENSION, --filename-extension FILENAME_EXTENSION
                        extension of generated file names (default: '.txt')
  --format {string,bits,tree,grammar,value,repr,none}
                        produce output(s) as string (default), as a bit
                        string, as a derivation tree, as a grammar, as a
                        Python value, in internal representation, or none
  --file-mode {text,binary,auto}
                        mode in which to open and write files (default is
                        'auto': 'binary' if grammar has bits or bytes, 'text'
                        otherwise)
  --validate            run internal consistency checks for debugging
  -o OUTPUT, --output OUTPUT
                        write output to OUTPUT (default: stdout)

general settings:
  -S START_SYMBOL, --start-symbol START_SYMBOL
                        the grammar start symbol (default: `<start>`)
  --warnings-are-errors
                        treat warnings as errors

algorithm settings:
  -N MAX_GENERATIONS, --max-generations MAX_GENERATIONS
                        the maximum number of generations to run the algorithm
  --population-size POPULATION_SIZE
                        the size of the population
  --elitism-rate ELITISM_RATE
                        the rate of individuals preserved in the next
                        generation
  --crossover-rate CROSSOVER_RATE
                        the rate of individuals that will undergo crossover
  --mutation-rate MUTATION_RATE
                        the rate of individuals that will undergo mutation
  --random-seed RANDOM_SEED
                        the random seed to use for the algorithm
  --destruction-rate DESTRUCTION_RATE
                        the rate of individuals that will be randomly
                        destroyed in every generation
  --max-repetition-rate MAX_REPETITION_RATE
                        rate at which the number of maximal repetitions should
                        be increased
  --max-repetitions MAX_REPETITIONS
                        Maximal value, the number of repetitions can be
                        increased to
  --max-node-rate MAX_NODE_RATE
                        rate at which the maximal number of nodes in a tree is
                        increased
  --max-nodes MAX_NODES
                        Maximal value, the number of nodes in a tree can be
                        increased to
  -n NUM_OUTPUTS, --num-outputs NUM_OUTPUTS, --desired-solutions NUM_OUTPUTS
                        the number of outputs to produce (default: 100)
```

(continues on next page)

(continued from previous page)

```
  --best-effort         produce a 'best effort' population (may not satisfy
                        all constraints)
  -i INITIAL_POPULATION, --initial-population INITIAL_POPULATION
                        directory or ZIP archive with initial population

command invocation settings:
  --input-method {stdin,filename}
                        when invoking COMMAND, choose whether Fandango input
                        will be passed as standard input (`stdin`) or as last
                        argument on the command line (`filename`) (default)
  command               command to be invoked with a Fandango input
  args                  the arguments of the command
```

Some of these options are very useful, such as `-o` and `-d`, which redirect the inputs generated towards individual files. You can also specify a *command* to be executed with the inputs Fandango generated.

We will go through these commands and options in due time. For now, let us get back to our specifications and actually fuzz with Fandango (page 25).

CHAPTER

NINE

FUZZING WITH FANDANGO

9.1 Creating a Name Database

Let us now come up with an input that is slightly more complex. We want to create a set of inputs with *names of persons* and their respective *age*. These two values would be *comma-separated*, such that typical inputs would look like this:

```
Alice Doe,27
John Smith,45
...
```

This makes the overall format of our input look like this:

```
<start> ::= <person_name> "," <age>
```

with `<age>` again being a sequence of digits, and a `<person>`'s name being defined as

```
<person_name> ::= <first_name> " " <last_name>
```

where both first and last name would be a sequence of letters - first, an uppercase letter, and then a sequence of lowercase letters. The full definition looks like this:

In Fandango specs, symbol names are formed as identifiers in Python - that is, they consist of letters, underscores, and digits.

Future Fandango versions will have shortcuts for specifying character ranges.

```
<start> ::= <person_name> "," <age>
<person_name> ::= <first_name> " " <last_name>
<first_name> ::= <name>
<last_name> ::= <name>
<name> ::= <ascii_uppercase_letter><ascii_lowercase_letter>+
<age> ::= <digit>+
```

The symbols `<ascii_uppercase_letter>`, `<ascii_lowercase_letter>`, and `<digits>` are predefined in the *Fandango Standard Library* (page 145); they are defined exactly as would be expected.

Create or download a file `persons.fan` and run Fandango on it:

```
$ fandango fuzz -f persons.fan -n 10
```

Your output will look like this:

```
Onwdgb Me,05789
Cvivd Qhibda,04
Ao Bqs,41408
Ch Xnajdo,032
Fjx Ops,5
Pduycd Qs,997
Owavk Jvsiu,63
Sjkh Aak,9
Dcliro Bzs,30
Bi Ntcg,298
```

Such random names are a typical result of *fuzzing* – that is, testing with randomly generated values. How do we get these into a program under test?

9.2 Feeding Inputs Into Programs

So far, we have had Fandango simply *output* the strings it generates. This is nice for understanding and debugging, but not necessarily useful for processing these inputs further. Fandango provides a number of means to help you process its output.

9.3 Feed Inputs into a Single file

Fandango has a `-o` option that directs its output into a single file. To store the generated strings in a file named `persons.txt`, use `-o persons.txt`:

```
$ fandango fuzz -f persons.fan -n 10 -o persons.txt
```

By default, the generated strings are separated by newlines. With the `-s` option, you can define an alternate separator. To have the outputs generated separated by `:`, for instance, use

```
$ fandango fuzz -f persons.fan -n 10 -o persons.txt -s ':'
```

9.4 Feed Inputs into Individual Files

With the `-d` option, Fandango can store its strings into individual files in the directory given after `-d`. They can then be picked up for post-processing. To store all outputs in a directory named `persons`, for instance, use `-d persons`:

```
$ fandango fuzz -f persons.fan -n 10 -d persons
```

Fandango will create the directory and store the inputs as `Fandango-0001.txt`, `Fandango-0002.txt`, etc.

Note

If the directory is already present, Fandango will use that directory.

Warning

Fandango will overwrite files `Fandango-0001.txt`, `Fandango-0002.txt`, etc., but leave other files in the directory unchanged.

If you want a different file extension (for instance, because `.txt` is not suitable), Fandango provides a `--filename-extension` option to set a different one.

9.5 Invoking Programs Directly

You can have Fandango invoke *programs directly*, and have Fandango feed them the generated inputs. The programs are specified as arguments on the command line.

There are two ways to pass the input into programs, on the command line and via standard input.

9.5.1 Passing Inputs on the Command Line

When Fandango invokes programs, you can pass input as a file name as last argument on the program's command line. This is the default, and can also be obtained with the `--input-method=filename` option.

For instance, to test the `wc` program with its own `-c` option and the inputs Fandango generates, use

```
$ fandango fuzz -f persons.fan -n 10 wc -c
```

The `wc` program is then invoked as `wc -c FILE_1`, `wc -c FILE_2`, etc., where each `FILE` contains an individual input from Fandango.

9.5.2 Passing Inputs via Standard Input

As an alternative, Fandango can pass inputs via standard input. For this, use the `--input-method=stdin` option.

For instance, to test the `cat` program with its own `-n` option and the inputs Fandango generates, use

```
$ fandango fuzz -f persons.fan -n 10 --input-method=stdin cat -n
```

The `cat` program is then invoked repeatedly, each time passing a new Fandango-generated input as its standard input.

9.6 Executable `.fan` files

On a Unix system, you can turn a `.fan` file into an *executable file* by placing a line

```
#!/usr/bin/env fandango fuzz -f
```

at the top. If you set its "executable" flag with `chmod +x FILE`, you can then directly execute the `.fan` file as a command as if it were prefixed by `fandango fuzz -f`.

As an example, let us create a file `fuzz-persons.fan`:

```
$ (echo '#!/usr/bin/env fandango fuzz -f'; cat persons.fan) > fuzz-persons.fan
$ chmod +x fuzz-persons.fan
```

You can now invoke the file, even with extra arguments:

```
$ ./fuzz-persons.fan -n 1
```

```
Tlnxv Paya,0675
```

CHAPTER TEN

SOME FUZZING STRATEGIES

The idea of fuzzing is to have inputs that are *out of the ordinary*, so we can detect errors in input parsers and beyond. But let us again have a look at the inputs we generated so far:

```
$ fandango fuzz -f persons.fan -n 10
```

```
Zsj Wytdd,52194
Qwzqsx Um,2956
Yzrr Ujtpng,716
Nali Qga,6
Upo Ktbkvl,3924
Tqdde Slvad,171
Zeroti Rutxs,58
Dkmx Ksacvh,610
Jfnhzh Ypphpg,77
Gjomc Jsvufb,2
```

Despite clearly looking non-natural to humans, the strings we generated so far are unlikely to trigger errors in a program, because programs typically treat all letters equally. So let us bring a bit more *weirdness* into our inputs.

Danger

Don't feed such fuzz inputs into other people's systems. In most countries, even *trying* will get you into jail.

10.1 Looooooong Inputs

One way to increase the probability of detecting bugs is to test with *long* inputs. While processing data, many programs have limited storage for individual data fields, and thus need to cope with inputs that exceed this storage space. If programmers do not care for such long inputs, serious errors may follow. So-called *buffer overflows* have long been among the most dangerous vulnerabilities, as they can be exploited to gain unauthorized access to systems.

We can easily create long inputs by specifying the *number of repetitions* in our grammar:

- Appending `{N}` to a symbol, where `N` is a number, makes Fandango repeat this symbol `N` times.
- Appending `{N,M}` to a symbol makes Fandango repeat this symbol between `N` and `M` times.
- Appending `{N,}` to a symbol makes Fandango repeat this symbol at least `N` times.

The `+`, `*`, and `?` suffixes are actually equivalent to `{1,}`, `{0,}`, and `{0,1}`, respectively.

If, for instance, we want the above names to be 100 characters long, we can set up a new rule for `<name>`

```
<name> ::= <ascii_uppercase_letter><ascii_lowercase_letter>{99}
```

and the lowercase letters will be repeated 99 times.

This is the effect of this rule:

```
Vqmtiaacrotwdktpwpaizhfuqbgspljtdestbhqefjmykokhhzkycsgoxucuwasoqosaxomfmoilrvysavzvphzgwvnbpvedi
 ↪Mkzynnrkfocabayhvffjmwxwhhyahgrdwnmlgrbiwaukfuscgwlgkprvejugdawzrxsrhfumdhsgjwfbjrultztwxttxhks
 ↪09
Swunkrsbrknoohqjedschysonkuoutedymwkgsvndacygoypwmdhbznmdqoooitjrrgxwxhdcotdesltdexmrjxgoolrsscz
 ↪Goukpsksnvcigvxbedokxjegrbsixmmewjwuvjndcirjvhagvgkylfsxedmgrlggbmjxcuirpdfmxxiouxdblltcylelmgx
 ↪74
Cfyvnxsimhffkmivjhyavoqdfeshezdrjpqdgubtrgzuaohsbgomcinxhrepzdpifaqybknvgpyzddsgpvmvreotdiefxwtp
 ↪Dofpqwqekndypgnlprzuhuiegunxrsnsojpgwhziuevlkdiwaladwowfbftycjdpbnncwmcnbtitgcxlfwuusqjcowcxspt
 ↪15
Kkmcmteuuqqtqehnomyqegxslsvynvrydpmlacfzlytcqtgqlrwmerjaayjgoxfiklruzwbhkdqxlousrtygjcgekzsdpodqh
 ↪Nnsvopxqookponsdbnxmyezzluxrythntsrwtcmvmlxowaorcqynxinqtzzsxbhzgehfmyvcthwdvjhprnmaltzghjomeae
 ↪8
Aflpsmfdyisjutohomembjveutfqrxhuernlkjyeprrpsdlmpiaiuuxejznzsntyiubdxdcronwkukcitlifhiwknklfbrnot
 ↪Qigotnjlmpipoenbfzhepgazeybitnimppusslodxhcaphmguelirjwjxxbjlrhhbrntuosjwibicaechszxqhvojyiwzry
 ↪0
Hwkfopwouieogvfkpyupahtotbpomsmiskmifhefuxbsbmjguponzrxlrhjwqtecccgbyfttettqayoiggjsruuvflymnzqes
 ↪Opxwevbhwoodtnjozjjkhnsircewnetyvszdtzlxdlbkyuiepwjyofwmzgtkdapzwspiuqvvtiripilxtydpyeyolxrsixl
 ↪57
Qydlojkduqolkxhlwvdpunszsblzeghwryfhpbkkkzblbojbojjbuayhvghcepkftwflmiukrsracmzpaamqoaddmmiyeesei
 ↪Uryokksvmypxxycnjezedfhsndkeceuwmcfccxsctsuhmvndndwlhzqyqvmvzvdebmiftuzexaeqniriupnmxgmvuhlodux
 ↪28
Leggizwhdizjufphdqfspiirlfkgwskkpdjqejsfnqeeopbjsuqvfmnaatixixlgmvhidfxkrfjpaijzlyzhgtewywpnndyxt
 ↪Aptjgkbxemazyssrnxntloahjnlakeviakjbdpkpxzzzrykmqhpgagggokzywrevtflhrrmxojnfeefjiihdrgpjiflneuf
 ↪33
Cevozjsnbohhcpbrqhdwclkdhcqccsssutyqvosinqrbywohlxecpyvstnayaihzqwclvfxicbnezzclgagsoimeiejntcdsi
 ↪Qxxxfbbmvhvznxkmoptrodgurfknajysttmzamfovmvvdwznnszpbuimqjezfkjburqchbfwvzusnyoewhiexspbdphmwvt
 ↪65
Ensidlstcgsqzsmtogbqildxmxvcjgpbrqtaqywauixzctzdzbjvgtqqtcbreunorcxfukzcfyujtmtblrsbeumdfwstvjghs
 ↪Obsqyfxwsdohedcinvsolxqgukqsrhaiyxgrxvdmvznmaqucosaupdunwccadigquezxgewrdktgvvrrkgpjbegtjnatfsb
 ↪3
```

We see that the names are now much longer. For real-world fuzzing, we may try even longer fields (say, 1,000) to test the limits of our system.

Note

Programmers often make "off-by-one" errors, so if an input is specified to have at most `N` characters, you should test this exact boundary - say, by giving `N` and `N+1` characters.

Danger

Don't try this with other people's systems.

10.2 Unusual Inputs

Having “weird” inputs also applies to numerical values. Think about our “age” field, for instance. What happens if we have a person with a negative age?

Note that the grammar already can produce ages way above 100.

Try it yourself and modify `persons.fan` such that it can also produce negative numbers, as in

```
Ewx Sbcg,-021
Ubmai Ei,7
Qqv Xb,269
Apedg Wx,-3
Ljigu Shyggk,2
Xokhwl Rfcuvw,89
Cxml Gepgyk,045
Ax Pu,-796
Gbfli Crdf,-187
Dlgnru Atdj,-48
```

Did you succeed? Compare your answer against the solution below.

Solution

You can, for instance, change the `<age>` rule such that it introduces an alternative for negative numbers:

```
<age> ::= <digit>+ | "-" <digit>+
```

Another way to do it is to use the `?` modifier, which indicates an optional symbol:

```
<age> ::= "-"? <digit>+
```

Other kinds of unusual inputs would be character sets that are out of the ordinary - for instance, Chinese or Hebrew characters - or plain Latin characters if your system expects Chinese names. A simple Emoji, for instance, could be enough to cause the system to fail.

In Python, try `float('nan') * float('inf')` and other variations.

For numbers, besides being out of range, there are a few constants that are interesting. Some common number parsers and converters accept values such as `Inf` (infinity) or `NaN` (not a number) as floating-point values. These actually *are* valid and have special rules – anything multiplied with `Inf` also becomes infinitely large (`Inf` times zero is zero, though); and any operation involving a `NaN` becomes `NaN`.

Fortunately, number converters typically treat `NaN` as invalid.

Imagine what happens if we manage to place a `NaN` value in a database? Any computation involving this value would also become a `NaN`, so in our example, the average age of persons would become `NaN`. The `NaN` could even go viral across Excel sheets, companies, shareholder reports, and eventually the stock market. Luckily, programs are prepared against that - or are they?

Danger

Don't try this with other people's systems.

10.3 String Injections

Another kind of attack is to insert special *strings* into the input – strings that would be interpreted by the program not as data, but as *commands*. A typical example for this is a *SQL injection*. Many programs use the SQL, the *structured query language*, as a means to interact with databases. A command such as

```
INSERT INTO CUSTOMERS VALUES ('John Smith', 34);
```

would be used to save the values `John Smith` (name) and `34` (age) in the `CUSTOMERS` table.

A SQL injection uses specially formed strings and values to subvert these commands. If our "age" value, for instance, is not `34`, but, say

```
34); CREATE TABLE PWNED (Phone CHAR(20)); --
```

then creating the above `INSERT` command with this special "age" could result in the following command:

Two dashes (`--`) start a comment in SQL.

```
INSERT INTO CUSTOMERS VALUES ('John Smith', 34); CREATE TABLE PWNED (Phone CHAR(20));␣
↪--);
```

and suddenly, we have "injected" a new command that will alter the database by adding a `PWNED` table.

How would one do this with a grammar? Well, for the above, it suffices to have one more alternative to the `<age>` rule.

Solution

Here's how one could change the `<age>` rule:

```
<age> ::= <digit>+ | "-" <digit>+
        | <digit>+ "); CREATE TABLE PWNED (Phone CHAR(20)); --"
```

Try adding such alternatives to *all* data fields processed by a system; feed the Fandango-generated inputs to it; and if you then find a `PWNED` table on your system, you know that you have a vulnerability.

Danger

Don’t try this with other people’s systems.

CHAPTER

ELEVEN

SHAPING INPUTS WITH CONSTRAINTS

11.1 The Limits of Grammars

So far, all the operations we have performed on our data were *syntax-oriented* - that is, we could shape the format and structure of our values, but not their *semantics* - that is, their actual meaning. Consider our *Person/Age dataset from the previous section* (page 25). What would we do if we want the “age” in a specific range? Or we want it to be odd? Or we want the age to be distributed in a certain way?

All of this refers to *context-free grammars*, which are the ones Fandango uses.

Some of these can be obtained by altering the grammar - limiting the age to two digits at most, for instance, will keep the value below 100 - but others cannot. Properties that cannot be expressed in a grammar are called *semantic properties* - in contrast to *syntactical properties*, which is precisely what grammars are good for.

11.2 Specifying Constraints

Fandango solves this problem through a pretty unique feature: It allows users to specify *constraints* which inputs have to satisfy. These thus narrow down the set of possible inputs.

Constraints are *predicates over grammar symbols*. Essentially, you write a Boolean expression, using grammar symbols (in `<...>`) to refer to individual elements of the input.

As an example, consider this Fandango constraint:

```
int(<age>) < 50
```

This constraint takes the `<age>` element from the input and converts it into an integer (all symbols are strings in the first place). Inputs are produced only if the resulting value is less than 50.

We can add such constraints to any .fan file, say the `persons.fan` file from the previous section. Constraints are preceded by a keyword `where`. So the line we add reads

```
where int(<age>) < 50
```

and the full `persons.fan` file reads

```
<start> ::= <person_name> "," <age>
<person_name> ::= <first_name> " " <last_name>
<first_name> ::= <name>
<last_name> ::= <name>
<name> ::= <ascii_uppercase_letter><ascii_lowercase_letter>+
<age> ::= <digit>+
where int(<age>) < 50
```

If we do this and run Fandango, we obtain a new set of inputs:

```
$ fandango fuzz -f persons.fan -n 10
```

```
Lx Exvfog,6
Illsur Uibds,31
Dqm Ynzxrs,005
Rvumy Eff,4
Plqc Ul,17
Mczlah Vugk,5
Qz Fjdzoj,8
Dc Xy,7
Ely Iary,6
Tvgcq Wzis,2
```

We see that all persons produced now indeed have an age of less than 50. Even if an age begins with `0`, it still represents a number below 50.

The language Fandango uses to express constraints is Python, so you can make use of arbitrary Python expressions. For instance, we can use Python Boolean operators (`and`, `or`, `not`) to request values in a range of 25-45:

Interestingly, having symbols in `<...>` does not conflict with the rest of Python syntax. Be sure, though, to leave spaces around < and > operators to avoid confusion.

```
25 <= int(<age>) and int(<age>) <= 45
```

and we obtain these inputs:

```
Mtff Ziyduq,27
Tz Eu,44
Cqivbq Aviay,32
```

```
Gvg Eveyc,40
Upi Nirkun,44
Nnswy Bl,43
Jel Cndny,40
Xaqlp Pnycgu,30
Zkx Pnycgu,30
Grrd Xgud,27
```

Start with `persons.fan` and add a constraint such that we generate people whose age is a multiple of 7, as in

```
Cftw Jbmz,5607
Eyddtp Okphmk,28
Kwkhbm Qhul,735
Pvoraz Vtjigy,07
Xxulil Ngmnq,63
Zbsuo Stgk,7
Xjoy Xjifzi,651
Flbis Zn,861
Akujn Dkm,11291
Kr Daxsrh,21357
```

(Hint: The modulo operator in Python is `%`).

Solution

This is not too hard. Simply add

```
where int(<age>) % 7 == 0
```

as a constraint.

11.3 Constraints and `DerivationTree` types

Whenever Fandango evaluates a constraint, such as

```
int(<age>) > 20
```

the type of `<age>` is actually not a string, but a `DerivationTree` object - *a tree representing the structure of the output.* (page 73). For now, you can use `DerivationTree` objects almost as if they were strings:

- You can *convert* them to other basic data types with (`int(<age>)`, `float(<age>)`, `str(<age>)`)
- You can invoke *string methods* on them (`.startswith('0'))
- You can *compare* them against each other (`<age_1> == <age_2>`) as well as against other strings (`<age> != "19"`)
- You can *print* them, using implicit string conversions (`print(<age>)`, which invokes `<age>.__str__()`)

One thing you *cannot* do, though, is *passing them directly as arguments to functions* that do not expect a `Derivation-Tree` type. This applies to the vast majority of Python functions.

Warning

If you want to pass a symbol as a function argument, convert it to the proper type (`int(<age>)`, `float(<age>)`, `str(<age>)`) first. Otherwise, you will likely raise an internal error in that very function.

Warning

On symbols, the `[...]` operator operates differently from strings - it returns a *subtree* (a substring) of the produced output: `<name>[0]` returns the `<first_name>` element, not the first character. If you want to access a character (or a range of characters) of a symbol, convert it into a string first, as in `str(<name>)[0]`.

We will learn more about derivation trees, `DerivationTree` types, and their operators in *Accessing Input Elements* (page 73).

11.4 Constraints on the Command Line

If you want to experiment with constraints, keeping on editing `.fan` files is a bit cumbersome. As an alternative, Fandango also allows to specify constraints on the command line. This is done with the `-c` (constraint) option, followed by the constraint expression (typically in quotes).

Starting with the original `persons.fan`, we can thus apply age constraints as follows:

```
$ fandango fuzz -f persons.fan -n 10 -c '25 <= int(<age>) and int(<age>) <= 45'
```

Constraints can be given multiple times, so the above can also be obtained as

```
$ fandango fuzz -f persons.fan -n 10 -c '25 <= int(<age>)' -c 'int(<age>) <= 45'
```

Note

On the command line, always put constraints in single quotes (`'...'`), as the angle brackets might otherwise be interpreted as I/O redirection.

When do constraints belong in a `.fan` file, and when on the command line? As a rule of thumb:

- If a constraint is *necessary* for obtaining valid input files (i.e. if the inputs would not be accepted otherwise), it belongs into the `.fan` file.
- If a constraint is *optional*, for instance for shaping inputs towards a particular goal, then it can also go on the command line.

11.5 How Fandango Solves Constraints

How does Fandango obtain these inputs? In a nutshell, Fandango is an *evolutionary* test generator:

1. It first uses the grammar to generate a *population* of inputs.
2. It then checks which individual inputs are *closest* in fulfilling the given constraints. For instance, for a constraint `int(<X>) == 100`, an input where `<X>` has a value of 90 is closer to fulfillment than one with value of, say 20.

Selecting the best inputs is also known as "survival of the fittest"

3. The best inputs are selected, the others are discarded.
4. Fandango then generates new *offspring* by *mutating* the remaining inputs, recomputing parts according to grammar rules. It can also exchange parts with those from other inputs; this is called *crossover*.
5. Fandango then repeats Steps 2-4 until all inputs satisfy the constraints.

All of this happens within Fandango, which runs through these steps with high speed. The `-v` option (verbose) produces some info on how the algorithm progresses:

```
$ fandango -v fuzz -f persons.fan -n 10 -c 'int(<age>) % 7 == 0'
```

```
fandango:INFO: ---------- Parsing FANDANGO content ----------
fandango:INFO: <stdlib>: loading cached spec from /Users/zeller/Library/Caches/
↪Fandango/c6d5b8d821f36cb11de70ec76c7a9f7fbe76431d76b22c2c0ad1c1d519a9e1a0.pickle
fandango:INFO: persons.fan: loading cached spec from /Users/zeller/Library/Caches/
↪Fandango/b414727fa16fea7721cf90f0cb6eac30d01a9c9c39c671088a7c3e894a1c0083.pickle
fandango:INFO: int(<age>) % 7 == 0: loading cached spec from /Users/zeller/Library/
↪Caches/Fandango/a753104490d484ad02278019279173c89eebe7fe1eab0add655483b140a71aef.
↪pickle
fandango:INFO: File mode: text
fandango:INFO: ---------- Initializing FANDANGO algorithm ----------
fandango:INFO: Generating initial population (size: 100)...
```

```
fandango:INFO: Initial population generated in 0.03 seconds
fandango:INFO: ---------- Starting evolution ----------
```

(continues on next page)

(continued from previous page)

```
fandango:INFO: ---------- Evolution finished ----------
fandango:INFO: Perfect solutions found: (10)
fandango:INFO: Fitness of final population: 1.00
fandango:INFO: Time taken: 0.00 seconds
Ij Fznnn,14
Hbe Rdemz,8638
Uliz Jxx,23478
Axyjye Zofz,91
Aawlwk Bbmolj,07
Gm Gd,98
Oh Ubldj,8351
Nmza Eipm,28854
Kgjbwz Jhjpyy,56
Mc Mzzhqy,0
fandango:INFO: Parsing '/var/folders/n2/xd9445p97rb3xh7m1dfx8_4h0006ts/T/
↪tmpn6caowcz/fandango-0001.txt'
fandango:INFO: Parsing '/var/folders/n2/xd9445p97rb3xh7m1dfx8_4h0006ts/T/
↪tmpn6caowcz/fandango-0002.txt'
fandango:INFO: Parsing '/var/folders/n2/xd9445p97rb3xh7m1dfx8_4h0006ts/T/
↪tmpn6caowcz/fandango-0003.txt'
fandango:INFO: Parsing '/var/folders/n2/xd9445p97rb3xh7m1dfx8_4h0006ts/T/
↪tmpn6caowcz/fandango-0004.txt'
fandango:INFO: Parsing '/var/folders/n2/xd9445p97rb3xh7m1dfx8_4h0006ts/T/
↪tmpn6caowcz/fandango-0005.txt'
fandango:INFO: Parsing '/var/folders/n2/xd9445p97rb3xh7m1dfx8_4h0006ts/T/
↪tmpn6caowcz/fandango-0006.txt'
fandango:INFO: Parsing '/var/folders/n2/xd9445p97rb3xh7m1dfx8_4h0006ts/T/
↪tmpn6caowcz/fandango-0007.txt'
fandango:INFO: Parsing '/var/folders/n2/xd9445p97rb3xh7m1dfx8_4h0006ts/T/
↪tmpn6caowcz/fandango-0008.txt'
fandango:INFO: Parsing '/var/folders/n2/xd9445p97rb3xh7m1dfx8_4h0006ts/T/
↪tmpn6caowcz/fandango-0009.txt'
fandango:INFO: Parsing '/var/folders/n2/xd9445p97rb3xh7m1dfx8_4h0006ts/T/
↪tmpn6caowcz/fandango-0010.txt'
```

Note

The `-v` option comes right after `fandango` (and not after `fandango fuzz`), as `-v` affects all commands (and not just `fuzz`).

11.6 When Constraints Cannot be Solved

Normally, Fandango continues evolving the population until all inputs satisfy the constraints. Some constraints, however, may be difficult or even impossible to solve. After a maximum number of generations (which can be set using `-N`), Fandango stops and produces the inputs it has generated so far. We can see this if we specify `False` as a constraint:

The `-N` option limits the number of generations - the default is 500.

```
$ fandango -v fuzz -f persons.fan -n 10 -c 'False' -N 50
```

```
fandango:INFO: ---------- Parsing FANDANGO content ----------
fandango:INFO: <stdlib>: loading cached spec from /Users/zeller/Library/Caches/
↪Fandango/c6d5b8d821f36cb11de70ec76c7a9f7fbe76431d76b22c2c0ad1c1d519a9e1a0.pickle
fandango:INFO: persons.fan: loading cached spec from /Users/zeller/Library/Caches/
↪Fandango/b414727fa16fea7721cf90f0cb6eac30d01a9c9c39c671088a7c3e894a1c0083.pickle
fandango:INFO: False: loading cached spec from /Users/zeller/Library/Caches/
↪Fandango/935a24ad0df474cb7a33185d32dd87047a14ee147ef83fa4320c0e0083ba48d4.pickle
fandango:INFO: File mode: text
fandango:INFO: ---------- Initializing FANDANGO algorithm ----------
fandango:INFO: Generating initial population (size: 100)...
```

```
fandango:INFO: Initial population generated in 0.03 seconds
fandango:INFO: ---------- Starting evolution ----------
fandango:INFO: Generation 1 - Fitness: 0.02 - #solutions found: 0
```

```
fandango:INFO: Generation 1: Increasing mutation rate from 0.20 to 0.22
fandango:INFO: Generation 1: Increasing crossover rate from 0.80 to 0.84
fandango:INFO: Generation 1: Increasing MAX_REPETITION from 5 to 7
fandango:INFO: Generation 1: Increasing Fuzzing Budget from 50 to 75
fandango:INFO: Generation 1 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 2 - Fitness: 0.00 - #solutions found: 0
fandango:INFO: Generation 2: Increasing mutation rate from 0.22 to 0.24
fandango:INFO: Generation 2: Increasing crossover rate from 0.84 to 0.88
fandango:INFO: Generation 2: Increasing MAX_REPETITION from 7 to 10
fandango:INFO: Generation 2: Increasing Fuzzing Budget from 75 to 112
fandango:INFO: Generation 2 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 3 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 3: Increasing mutation rate from 0.24 to 0.27
fandango:INFO: Generation 3: Increasing crossover rate from 0.88 to 0.93
fandango:INFO: Generation 3: Increasing MAX_REPETITION from 10 to 15
fandango:INFO: Generation 3: Increasing Fuzzing Budget from 112 to 168
fandango:INFO: Generation 3 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 4 - Fitness: 0.00 - #solutions found: 0
fandango:INFO: Generation 4: Increasing mutation rate from 0.27 to 0.29
fandango:INFO: Generation 4: Increasing crossover rate from 0.93 to 0.97
fandango:INFO: Generation 4: Increasing MAX_REPETITION from 15 to 22
fandango:INFO: Generation 4: Increasing Fuzzing Budget from 168 to 200
fandango:INFO: Generation 4 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 5 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 5: Increasing mutation rate from 0.29 to 0.32
fandango:INFO: Generation 5: Increasing crossover rate from 0.97 to 1.00
fandango:INFO: Generation 5: Increasing MAX_REPETITION from 22 to 33
fandango:INFO: Generation 5 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 6 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 6: Increasing mutation rate from 0.32 to 0.35
fandango:INFO: Generation 6: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 6: Increasing MAX_REPETITION from 33 to 49
fandango:INFO: Generation 6 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 7 - Fitness: 0.00 - #solutions found: 0
fandango:INFO: Generation 7: Increasing mutation rate from 0.35 to 0.39
fandango:INFO: Generation 7: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 7: Increasing MAX_REPETITION from 49 to 73
fandango:INFO: Generation 7 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 8 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 8: Increasing mutation rate from 0.39 to 0.43
fandango:INFO: Generation 8: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 8: Increasing MAX_REPETITION from 73 to 109
fandango:INFO: Generation 8 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 9 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 9: Increasing mutation rate from 0.43 to 0.47
fandango:INFO: Generation 9: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 9: Increasing MAX_REPETITION from 109 to 163
fandango:INFO: Generation 9 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 10 - Fitness: 0.00 - #solutions found: 0
fandango:INFO: Generation 10: Increasing mutation rate from 0.47 to 0.52
fandango:INFO: Generation 10: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 10: Increasing MAX_REPETITION from 163 to 244
```

```
fandango:INFO: Generation 10 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 11 - Fitness: 0.00 - #solutions found: 0
fandango:INFO: Generation 11: Increasing mutation rate from 0.52 to 0.57
fandango:INFO: Generation 11: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 11: Increasing MAX_REPETITION from 244 to 366
fandango:INFO: Generation 11 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 12 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 12: Increasing mutation rate from 0.57 to 0.63
fandango:INFO: Generation 12: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 12: Increasing MAX_REPETITION from 366 to 549
fandango:INFO: Generation 12 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 13 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 13: Increasing mutation rate from 0.63 to 0.69
fandango:INFO: Generation 13: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 13: Increasing MAX_REPETITION from 549 to 823
fandango:INFO: Generation 13 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 14 - Fitness: 0.00 - #solutions found: 0
fandango:INFO: Generation 14: Increasing mutation rate from 0.69 to 0.76
fandango:INFO: Generation 14: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 14: Increasing MAX_REPETITION from 823 to 1234
```

(continues on next page)

(continued from previous page)

```
fandango:INFO: Generation 14 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 15 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 15: Increasing mutation rate from 0.76 to 0.84
fandango:INFO: Generation 15: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 15: Increasing MAX_REPETITION from 1234 to 1851
fandango:INFO: Generation 15 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 16 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 16: Increasing mutation rate from 0.84 to 0.92
fandango:INFO: Generation 16: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 16: Increasing MAX_REPETITION from 1851 to 2776
fandango:INFO: Generation 16 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 17 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 17: Increasing mutation rate from 0.92 to 1.00
fandango:INFO: Generation 17: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 17: Increasing MAX_REPETITION from 2776 to 4164
fandango:INFO: Generation 17 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 18 - Fitness: 0.00 - #solutions found: 0
fandango:INFO: Generation 18: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 18: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 18: Increasing MAX_REPETITION from 4164 to 6246
```

```
fandango:INFO: Generation 18 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 19 - Fitness: 0.00 - #solutions found: 0
fandango:INFO: Generation 19: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 19: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 19: Increasing MAX_REPETITION from 6246 to 9369
```

```
fandango:INFO: Generation 19 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 20 - Fitness: 0.00 - #solutions found: 0
fandango:INFO: Generation 20: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 20: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 20: Increasing MAX_REPETITION from 9369 to 14053
fandango:INFO: Generation 20 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 21 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 21: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 21: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 21: Increasing MAX_REPETITION from 14053 to 21079
fandango:INFO: Generation 21 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 22 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 22: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 22: Increasing crossover rate from 1.00 to 1.00
```

(continues on next page)

(continued from previous page)

```
fandango:INFO: Generation 22: Increasing MAX_REPETITION from 21079 to 31618
fandango:INFO: Generation 22 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 23 - Fitness: 0.00 - #solutions found: 0
fandango:INFO: Generation 23: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 23: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 23: Increasing MAX_REPETITION from 31618 to 47427
fandango:INFO: Generation 23 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 24 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 24: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 24: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 24: Increasing MAX_REPETITION from 47427 to 71140
fandango:INFO: Generation 24 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 25 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 25: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 25: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 25: Increasing MAX_REPETITION from 71140 to 106710
fandango:INFO: Generation 25 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 26 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 26: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 26: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 26: Increasing MAX_REPETITION from 106710 to 160065
fandango:INFO: Generation 26 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 27 - Fitness: 0.00 - #solutions found: 0
fandango:INFO: Generation 27: Increasing mutation rate from 1.00 to 1.00
```

```
fandango:INFO: Generation 27: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 27: Increasing MAX_REPETITION from 160065 to 240097
fandango:INFO: Generation 27 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 28 - Fitness: 0.00 - #solutions found: 0
fandango:INFO: Generation 28: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 28: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 28: Increasing MAX_REPETITION from 240097 to 360145
fandango:INFO: Generation 28 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 29 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 29: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 29: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 29: Increasing MAX_REPETITION from 360145 to 540217
fandango:INFO: Generation 29 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 30 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 30: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 30: Increasing crossover rate from 1.00 to 1.00
```

(continues on next page)

(continued from previous page)

```
fandango:INFO: Generation 30: Increasing MAX_REPETITION from 540217 to 810325
fandango:INFO: Generation 30 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 31 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 31: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 31: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 31: Increasing MAX_REPETITION from 810325 to 1215487
fandango:INFO: Generation 31 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 32 - Fitness: 0.00 - #solutions found: 0
fandango:INFO: Generation 32: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 32: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 32: Increasing MAX_REPETITION from 1215487 to 1823230
```

```
fandango:INFO: Generation 32 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 33 - Fitness: 0.00 - #solutions found: 0
fandango:INFO: Generation 33: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 33: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 33: Increasing MAX_REPETITION from 1823230 to 2734845
fandango:INFO: Generation 33 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 34 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 34: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 34: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 34: Increasing MAX_REPETITION from 2734845 to 4102267
fandango:INFO: Generation 34 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 35 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 35: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 35: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 35: Increasing MAX_REPETITION from 4102267 to 6153400
fandango:INFO: Generation 35 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 36 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 36: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 36: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 36: Increasing MAX_REPETITION from 6153400 to 9230100
fandango:INFO: Generation 36 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 37 - Fitness: 0.00 - #solutions found: 0
fandango:INFO: Generation 37: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 37: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 37: Increasing MAX_REPETITION from 9230100 to 13845150
fandango:INFO: Generation 37 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 38 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 38: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 38: Increasing crossover rate from 1.00 to 1.00
```

(continues on next page)

(continued from previous page)

```
fandango:INFO: Generation 38: Increasing MAX_REPETITION from 13845150 to 20767725
fandango:INFO: Generation 38 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 39 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 39: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 39: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 39: Increasing MAX_REPETITION from 20767725 to 31151587
fandango:INFO: Generation 39 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 40 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 40: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 40: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 40: Increasing MAX_REPETITION from 31151587 to 46727380
fandango:INFO: Generation 40 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 41 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 41: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 41: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 41: Increasing MAX_REPETITION from 46727380 to 70091070
fandango:INFO: Generation 41 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 42 - Fitness: 0.00 - #solutions found: 0
fandango:INFO: Generation 42: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 42: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 42: Increasing MAX_REPETITION from 70091070 to 105136605
fandango:INFO: Generation 42 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 43 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 43: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 43: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 43: Increasing MAX_REPETITION from 105136605 to 157704907
fandango:INFO: Generation 43 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 44 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 44: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 44: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 44: Increasing MAX_REPETITION from 157704907 to 236557360
fandango:INFO: Generation 44 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 45 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 45: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 45: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 45: Increasing MAX_REPETITION from 236557360 to 354836040
fandango:INFO: Generation 45 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 46 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 46: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 46: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 46: Increasing MAX_REPETITION from 354836040 to 532254060
fandango:INFO: Generation 46 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
 ↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 47 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 47: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 47: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 47: Increasing MAX_REPETITION from 532254060 to 798381090
fandango:INFO: Generation 47 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
 ↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 48 - Fitness: 0.00 - #solutions found: 0
fandango:INFO: Generation 48: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 48: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 48: Increasing MAX_REPETITION from 798381090 to␣
 ↪1197571635
```

```
fandango:INFO: Generation 48 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
 ↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 49 - Fitness: 0.00 - #solutions found: 0
fandango:INFO: Generation 49: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 49: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 49: Increasing MAX_REPETITION from 1197571635 to␣
 ↪1796357452
fandango:INFO: Generation 49 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
 ↪diversity: 0.02, Population size: 100
fandango:INFO: Generation 50 - Fitness: 0.00 - #solutions found: 0
```

```
fandango:INFO: Generation 50: Increasing mutation rate from 1.00 to 1.00
fandango:INFO: Generation 50: Increasing crossover rate from 1.00 to 1.00
fandango:INFO: Generation 50: Increasing MAX_REPETITION from 1796357452 to␣
 ↪2694536178
fandango:INFO: Generation 50 stats -- Best fitness: 0.00, Avg fitness: 0.00, Avg␣
 ↪diversity: 0.02, Population size: 100
fandango:INFO: ---------- Evolution finished ----------
fandango:INFO: Perfect solutions found: (0)
fandango:INFO: Fitness of final population: 0.00
fandango:INFO: Time taken: 2.05 seconds
fandango:ERROR: Population did not converge to a perfect population
fandango:ERROR: Only found 0 perfect solutions, instead of the required 10
```

As you see, Fandango produces a population of zero. Of course, if the constraint is `False`, then there can be no success.

Tip

Fandango has a `--best-effort` option that allows you to still output the final population.

CHAPTER

TWELVE

THE FANDANGO SHELL

Sometimes, you may find cumbersome to invoke Fandango from the command line again and again. This is especially true when

- You want to explore the effects of different constraints.
- You want to explore the effects of different algorithm settings.

This is why Fandango offers a *shell*, in which

- you can enter commands directly at a Fandango prompt
- you can set and edit parameters *once* for future commands
- settings are preserved during your session
- your command history is preserved across sessions.

12.1 Invoking the Fandango Shell

Invoking the Fandango shell is easy. Just invoke Fandango without a command:

```
$ fandango
```

and you will be greeted by the Fandango prompt

```
(fandango)
```

12.2 Invoking Commands

At the `(fandango)` prompt, you can enter the same commands you already know from the command line, such as

```
(fandango) fuzz -f persons.fan -n 10
```

Hint

Use TAB to complete commands, options, and file names.

and you will get the same results:

```
Yjvtj Ycgqut,33
Vrxvm Sxk,297
Ftmfw Xxwh,6399
Ejva Iejr,11797
Frjopb Ghoeg,95
Vkpuu Vfrvw,364
Yvays Kyev,5578
Fqs El,43
Yafuy Etsfkb,65
Ikog Xj,9
```

There is an important advantage though: You can set (and edit) a *common environment* for all commands.

12.3 Setting a Common Environment

The `set` command (available only in the Fandango shell) allows you to specify resources and settings that are then applied to all later commands (notably, `fuzz`).

You can, for instance, specify a `.fan` file with

```
(fandango) set -f persons.fan -n 10
```

Note

A big advantage of the `set` command is that the `.fan` file is read only *once*, and then available for all later commands.

After options for resources and settings are set, you can omit them from later `fuzz` commands:

```
(fandango) fuzz
```

Note

If you give `fuzz` additional options, these temporarily *override* the settings given with `set` earlier. As an exception, *constraints* (`-c`) are *added* to those already set.

The options for the `set` command are roughly the same as for the `fuzz` command and include

- `.fan` files (`set -f FILE`)
- constraints (`set -c CONSTRAINT`)
- algorithm settings (`set --mutation-rate=0.2`)

To get a full list of options, try `help set`.

Note

Some command-specific options like `-o` or `-d` (controlling the output of the `fuzz` command) are not available for setting with `set`.

12.4 Retrieving Settings

To retrieve the current settings, simply enter `set`. This will list:

- the grammar and constraints of the current `.fan` file
- as well as any constraints or settings you have made

Here is an example:

```
(fandango) set -N 10
(fandango) set
<start> ::= <person_name> ',' <age>
<person_name> ::= <first_name> ' ' <last_name>
<first_name> ::= <name>
<last_name> ::= <name>
<name> ::= <ascii_uppercase_letter> <ascii_lowercase_letter>+
<age> ::= <digit>+
--max-generations=10
```

12.5 Resetting Settings

To reset all settings to default values, use the `reset` command:

```
(fandango) reset
```

This will

- clear all constraints defined with `set`
- reset all algorithm settings to their default value.

The current `.fan` file stays unchanged.

Attention

Loading a new `.fan` file also clears all `set` constraints.

12.6 Quotes and Escapes

The Fandango shell uses the same quoting conventions as a POSIX system shell. For instance, to set a constraint, place it in quotes:

```
(fandango) set -c 'int(<age>) < 30'
```

You can also escape individual characters with backslashes:

```
(fandango) set -c int(<age>)\ <\ 30
```

12.7 Editing Commands

The Fandango shell uses the GNU readline library. Therefore, you can

- Use cursor keys *left* and *right* to edit commands
- Use cursor keys *up* and *down* to scroll through history
- Use the *TAB* key to expand command names, options, and arguments.

The command history is saved in `~/.fandango_history`.

12.8 Invoking Commands from the Shell

In the Fandango shell, you can invoke *system commands* by prefixing them with `!`:

```
(fandango) !ls
LICENSE.md                docs                            requirements.txt
Makefile                language                     src
README.md                pyproject.toml                 tests
```

Hint

Use TAB to complete file names.

You can also invoke and evaluate *Python commands* by prefixing them with `/`:

```
(fandango) /import random
(fandango) /random.randint(10, 20)
11
```

If the command you enter has a value, the value is automatically printed.

Note

Invoking system and Python commands is only available when the input is a terminal.

12.9 Changing the Current Directory

To change the current directory, Fandango provides a built-in `cd` command:

The alternative `!cd` does not work, as this changes the directory of the invoked shell.

```
(fandango) cd docs/
```

Without arguments, `cd` switches to the home directory.

Attention

This is different from Windows, where `cd` reports the current directory.

12.10 Getting Shell Commands from a File

Instead of entering commands by keyboard, one can also have Fandango read in commands from a file or another command. This is done by redirecting the `fandango` standard input. To have Fandango read and execute commands from, say, `commands.txt`, use

```
$ fandango < commands.txt
```

Fandango can also process the commands issued from another program:

```
$ echo 'fuzz -n 10 -f persons.fan' | fandango
```

```
Dpmc Zdc,9239
Mb Sq,3
Tdr Tzufdq,681
Dtj Udiv,19028
Xnanuf Bghbo,37
```

```
Xggqcy Tmdj,01
Zxzedh Nppn,046
Ww Exnnvo,4
Uxcq Dedim,38
Yfmf Xoov,652
```

Hint

The input file can contain blank lines as well as comments prefixed with `#`.

Note

System commands (`!`) and Python commands (`/`) are not available when reading from a file.

12.11 Exiting the Fandango Shell

To exit the Fandango shell and return to the system command line, enter the command `exit`:

```
(fandango) exit
$
```

Entering an EOF (end-of-file) character, typically `Ctrl-D`, will also exit the shell.

```
(fandango) ^D
$
```

In the next section, we'll talk about *custom generators* (page 53).

CHAPTER

THIRTEEN

DATA GENERATORS AND FAKERS

Often, you don't want to generate *totally* random data; it suffices that *some* aspects of it are random. This naturally raises the question: Where can one get non-random, *natural* data from, and how can one integrate this into Fandango?

13.1 Augmenting Grammars with Data

The straightforward solution would be to simply extend our grammar with more natural data. In order to obtain more natural first and last names in our *ongoing names/age example* (page 25), for instance, we could simply extend the `persons.fan` rule

```
<first_name> ::= <name>
```

to

These are the given names on Pablo Picasso[9]'s birth certificate.

```
<first_name> ::= <name> | "Alice" | "Bob" | "Eve" | "Pablo Diego José Francisco de␣
↪Paula Juan Nepomuceno Cipriano de la Santísima Trinidad"
```

and extend the rule

```
<last_name> ::= <name>
```

to, say,

```
<last_name> ::= <name> | "Doe" | "Smith" | "Ruiz Picasso"
```

then we can have Fandango create names such as

```
Eve Ruiz Picasso,7
Pablo Diego José Francisco de Paula Juan Nepomuceno Cipriano de la Santísima␣
↪Trinidad Smith,8704
Eve Smith,7614
Bob Ruiz Picasso,32
Bob Ruiz Picasso,41
Alice Ruiz Picasso,7
```

[9] https://en.wikipedia.org/wiki/Pablo_Picasso

```
Bob Smith,511
Alice Doe,26270
Bob Nyvkh,33
Alice Ruiz Picasso,33
```

Note that we still get a few “random” names; this comes as specified by our rules. By default, Fandango picks each alternative with equal likelihood, so there is a 20% chance for the first name and a 25% chance for the last name to be completely random.

Note

Future Fandango versions will have means to control these likelihoods.

13.2 Using Fakers

Frequently, there already are data sources available that you'd like to reuse – and converting each of their elements into a grammar alternative is inconvenient. That is why Fandango allows you to *specify a data source as part of the grammar* - as a *Python function* that supplies the respective value. Let us illustrate this with an example.

The Python `faker` module is a great source of “natural” data, providing “fake” data for names, addresses, credit card numbers, and more.

Here's an example of how to use it:

```python
from faker import Faker
fake = Faker()
for i in range(10):
  print(fake.first_name())
```

```
Shane
Patrick
Jason
Joshua
Samuel
Andrew
Heather
John
Linda
Kristen
```

Have a look at the `faker` documentation[10] to see all the fake data it can produce. The methods `first_name()` and `last_name()` are what we need. The idea is to extend the `<first_name>` and `<last_name>` rules such that they can draw on the `faker` functions. To do so, in Fandango, you can simply extend the grammar as follows:

A generator applies to *all* alternatives, not just the last one.

```
<first_name> ::= <name> := fake.first_name()
```

[10] https://faker.readthedocs.io/

`:=` is the assignment operator in several programming languages; in Python, it can be used to assign values within expressions.

The *generator* `:= EXPR` *assigns* the value produced by the expression `EXPR` (in our case, `fake.first_name()`) to the symbol on the left-hand side of the rule (in our case, `<first_name>`).

Caution

Whatever value the generator returns, it must be *parseable* by at least one of the alternatives in the rule. Our example works because `<first_name>` matches the format of `fake.first_name()`.

Tip

If your generator returns a string, a "match-all" rule such as

```
<generated_string> ::= <char>* := generator()
```

will fit all possible string values returned by `generator()`.

We can do the same for the last name, too; and then this is the full Fandango spec persons-faker.fan:

```
from faker import Faker
fake = Faker()

include('persons.fan')

<first_name> ::= <name> := fake.first_name()
<last_name> ::= <name> := fake.last_name()
```

Note

The *Fandango `include()` function* (page 167) *includes* the Fandango definitions of the given file. This way, we need not repeat the definitions from `persons.fan` and only focus on the differences.

Note

Python code (from Python files) that you use in a generator (or in a constraint, for that matter) needs to be imported. Use the Python `import` features to do that.

Attention

`include(FILE)` is for Fandango files, `import MODULE` is for Python modules.

This is what the output of the above spec looks like:

```
Spencer Wilson,62
Robert Murphy,36
Ebony Williams,15
Aaron Reynolds,06
Daniel Goodwin,00
Eric Brown,30
Douglas Jackson,3
David Vega,51
Brian Howell,63
Crystal Anderson,99
```

You see that all first and last names now stem from the Faker library.

13.3 Number Generators

In the above output, the "age" fields are still very random, though. With generators, we can achieve much more natural distributions.

After importing the Python `random` module:

```
import random
```

we can make use of dozens of random number functions[11] to use as generators. For instance, `random.randint(A, B)` return a random integer n such that $A \leq n \leq B$ holds. To obtain a range of ages between 25 and 35, we can thus write:

```
<age> ::= <digit>+ := str(random.randint(25, 35));
```

Warning

All Fandango generators must return strings or byte strings.

- Use `str(N)` to convert a number N into a string
- Use `bytes([N])` to convert numbers N into bytes.

The resulting Fandango spec file produces the desired range of ages:

```
Steven Rodriguez,29
Robert Harrington,35
James Diaz,34
Adam Martinez,34
Richard Jensen,32
Jason Roman,34
Isabella Hunter,26
Kenneth Davis,28
Ronald Thomas,33
Julie Hull,33
```

We can also create a Gaussian (normal) distribution this way:

[11] https://docs.python.org/3/library/random.html

```
<age> ::= <digit>+ := str(int(random.gauss(35)));
```

`random.gauss()` returns floating point numbers. However, the final value must fit the given symbol rules (in our case, `<digit>+`), so we convert the age into an integer (`int()`).

These are the ages we get this way:

```
Jennifer Novak,36
Sarah Randolph,34
Robert Stephens,35
Clarence Tucker,35
Jennifer Thornton,35
Bryan Sampson,34
Monica Weaver,35
Christine Cook,33
Emily Luna,34
Michael Santiago,34
```

In sec:distributions, we will introduce more ways to obtain specific distributions.

13.4 Generators and Random Productions

In testing, you want to have a good balance between *common* and *uncommon* inputs:

- Common inputs are important because they represent the typical usage, and you don't want your program to fail there;
- Uncommon inputs are important because they uncover bugs you may *not* find during alpha or beta testing, and thus avoid latent bugs (and vulnerabilities!) slipping into production.

We can easily achieve such a mix by adding rules such as

```
<first_name> ::= <name> | <natural_name>
<natural_name> ::= <name> := fake.first_name()
```

With this, both random names (`<name>`) and natural names (`<natural_name>`) will have a chance of 50% to be produced:

```
Qfkqz Frey,8069
Dana Douglas,10
Whitney Howard,80
Mandy Frazier,28
Matthew Gomez,65
Tjp Brown,069
Austin Boyd,357
Snzukp Cook,693
Nd Durham,619
Penny Woods,598
```

13.5 Combining Generators and Constraints

When using a generator, why does one still have to specify the format of the data, say `<name>`? This is so for two reasons:

To allow access to the elements of the generator output, Fandango *parses* the output according to the symbol rules.

1. It allows the Fandango spec to be used for *parsing* existing data, and consequently, *mutating* it;
2. It allows additional constraints (page 35) to be applied on the generator result and its elements.

In our example, the latter can be used to further narrow down the set of names. If we want all last names to start with an `S`, for instance, we can invoke Fandango as

Grammar symbols `<...>` support all Python string methods[12].

```
$ fandango fuzz -f persons-faker.fan -c '<last_name>.startswith("S")' -n 10
```

and we get

```
Todd Simmons,5
Denise Smith,5
Rhonda Smith,30
Susan Simpson,23
David Solomon,19
Alicia Scott,85
Diana Scott,819
Melanie Sanchez,9
Angel Spence,47
Andrew Stephens,65
```

13.6 When to use Generators, and when Constraints

One might assume that instead of a generator, one could also use a *constraint* to achieve the same effect. So, couldn't one simply add a constraint that says

```
<first_name> == fake.first_name()
```

In case this should work, this is only through some internal Fandango optimization that directly assigns computed values to symbols.

[12] https://docs.python.org/3/library/stdtypes.html#string-methods

Unfortunately, this does not work.

The reason is that the faker returns *a different value* every time it is invoked, making it hard for Fandango to solve the constraint. Remember that Fandango solves constraints by applying mutations to a population, getting closer to the target with each iteration. If the target keeps on changing, the algorithm will lose guidance and will not progress towards the solution.

Likewise, in contrast to our example in *Combining Generators and Constraints* (page 58), one may think about using a *constraint* to set a limit to a number, say:

```
$ fandango fuzz -f persons-faker.fan -c 'str(<last_name>).startswith("S")' -c 'int(
↪<age>) >= 25 and int(<age>) <= 35' -n 10
```

This would work:

```
Wendy Smith,32
Glenn Smith,25
Chad Smith,27
Daniel Smith,32
Jeffrey Smith,32
Ashley Skinner,35
Chad Smith,28
Wendy Smith,33
Glenn Smith,27
Chad Sanders,27
```

But while the values will fit the constraint, they will not be randomly distributed. This is because Fandango treats and generates them as *strings* (= sequences of digits), ignoring thur semantics as numerical values. To obtain well-distributed numbers from the beginning, use a generator.

1. If a value to be produced is *random*, it should be added via a *generator*.
2. If a value to be produced is *constant*, it can go into a *generator* or a *constraint*.
3. If a value to be produced must be *part of a valid input*, it should go into a *constraint*. (Constraints are checked during parsing *and* production.)

CHAPTER

FOURTEEN

REGULAR EXPRESSIONS

Although the Fandango grammars cover a wide range of input language features, there are situations where they may be a bit cumbersome to work with. Consider specifying *every digit except for zeros*: this requires you to enumerate all the other digits `1`, `2`, and so on. This is why Fandango also supports *regular expressions*, which allow you to use a concise syntax for character ranges, repeated characters and more. Specifying all digits from `1` to `9`, for instance, becomes the short regular expression `r'[1-9]'`.

14.1 About Regular Expressions

Regular expressions form a language on their own and come with several useful features. To get an introduction to the regular expressions Fandango uses, read the Python Regular Expression HOWTO[13] and check out the Python Regular Expression Syntax[14] for a complete reference.

In Fandango, regular expressions are used for two purposes:

- When *producing* inputs, a regular expression is instantiated into a random string that matches the expression.
- When *parsing* inputs, a regular expression is used to *parse* and *match* inputs.

14.2 Writing Regular Expressions

For Python aficionados: this is actually a Python "raw string"

In Fandango, a regular expression comes as a string, prefixed with a `r` character. To express that a digit can have the values `0` to `9`, instead of

```
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

you can write

```
<digit> ::= r'[0-9]'
```

which is much more concise.

Likewise, to match a sequence of characters that ends in `;`, you can write

[13] https://docs.python.org/3/howto/regex.html

[14] https://docs.python.org/3/library/re.html#regular-expression-syntax

```
<some_sequence> ::= r'[^;]+;'
```

Besides the `r` prefix indicating a regular expression, it also makes the string a *raw* string. This means that backslashes are treated as *literal characters*. The regular expression `\d`, for instance, matches a Unicode digit, which includes `[0-9]`, and also many other digit characters[15]. To include `\d` in a regular expression, write it *as is*; do not escape the backslash with another backslash (as you would do in a regular string):

The expression `r'\\d'` would actually match a backslash, followed by a `d` character.

```
<any_digit> ::= r'\d'
```

Warning

Be aware of the specific syntax of `r`-strings as it comes to backslashes.

One consequence of backslashes being interpreted literally is that you cannot escape quote characters in a regular expression. This causes a problem if you need two kinds of quotes (`"` and `'`) in the same regular expression – say, a rule that checks for forbidden characters.

However, encodings of the form `\xNN` are also interpreted by regular expressions. Hence, if you need quotes, you can use

- `\x22` instead of `"`
- `\x27` instead of `'`

Here is an example:

```
<forbidden_characters> ::= r'[\x22\x27;]'
```

14.3 Fine Points about Regular Expressions

For parsing inputs, Fandango uses the Python `re`[16] module for matching strings against regular expressions; for producing inputs, Fandango uses the Python `exrex`[17] module for generating strings that match regular expressions. All the `re` and `exrex` capabilities and limitations thus extend to Fandango.

[15] https://en.wikipedia.org/wiki/Numerals_in_Unicode
[16] https://docs.python.org/3/library/re.html
[17] https://github.com/asciimoo/exrex

14.3.1 Repetition Limits

Most notably, `exrex` imposes a *repetition limit* of 20 on generated strings that in principle can have arbitrary length; a `+` or `*` operator will not expand to more than 20 repetitions. Thus, a grammar `infinity.fan`

```
<start> ::= r"(abc)+"
```

that in principle, could produce arbitrary long sequences `abcabcabcabc...` will be limited to 20 repetitions at most:

```
$ fandango fuzz -f infinity.fan -n 10
```

```
fandango:WARNING: Could not generate a full population of unique individuals.␣
↪Population size reduced to 20.
abcabcabcabcabcabcabcabcabcabcabcabcabcabc
abcabc
abc
abcabcabc
abcabcabcabcabc
abcabcabcabc
abcabcabcabcabcabcabcabc
abcabcabcabcabcabcabcabcabcabcabcabcabcabcabc
abcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabc
abcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabc
```

To precisely control the number of repetitions, use the regular expression `{m,n}` construct, limiting the number of repetitions from `m` to `n`. Let us limit the number of repetitions to the range 1...5:

```
<start> ::= r"(abc){1,5}"
```

This is what we get:

```
$ fandango fuzz -f finity.fan -n 10
```

```
fandango:WARNING: Could not generate a full population of unique individuals.␣
↪Population size reduced to 5.
fandango:ERROR: Error during crossover: Sample larger than population or is␣
↪negative
fandango:ERROR: Error during crossover: Sample larger than population or is␣
↪negative
fandango:WARNING: Could not generate full unique new population, filling remaining␣
↪slots with duplicates.
abc
abcabcabcabc
abcabcabcabcabc
abcabcabc
abcabc
abcabc
abcabc
abcabc
abcabc
abcabcabcabc
abcabcabcabc
abcabcabcabc
abcabcabcabcabc
abcabcabc
abcabc
abcabcabcabc
```

(continues on next page)

(continued from previous page)

```
abcabcabcabcabc
abcabcabc
abcabcabc
abcabcabc
abcabcabcabc
abcabcabcabcabc
abcabcabcabcabc
abcabcabcabc
abc
abcabcabcabcabc
abc
abcabc
abc
abcabcabc
abcabcabc
abc
abc
abcabc
abcabc
abcabc
abcabcabcabc
abc
abcabcabc
abcabcabcabcabc
abcabcabcabc
abcabcabcabcabc
abcabcabcabc
abcabcabcabc
abcabcabc
abcabcabc
abc
abcabc
abc
abc
abcabcabcabcabc
abcabcabc
abcabcabcabc
abc
abcabcabc
abcabcabcabc
abcabcabcabcabc
abc
abcabcabcabc
abc
abcabcabcabc
abcabc
abcabcabcabc
abc
abcabc
abcabcabcabcabc
abcabcabc
abcabcabc
abcabcabc
abc
abcabcabcabc
abc
abcabcabcabcabc
```

(continues on next page)

(continued from previous page)

```
abc
abcabcabcabcabc
abcabcabcabc
abcabcabc
abcabc
abcabcabc
abcabcabcabcabc
abcabcabcabcabc
abcabcabc
abcabcabcabcabc
abc
abcabcabc
abcabc
abcabc
abcabcabcabcabc
abcabc
abc
abcabc
abcabcabcabcabc
abcabcabcabc
abcabcabcabc
abcabcabc
abcabcabc
abc
abc
abcabcabcabcabc
abc
```

Tip

Remember that *grammars* also have operators `+`, `*`, `?`, and `{N,M}` which apply to the preceding grammar element, and work like their *regular expression* counterparts. Using these, we could also write the above as

```
<start> ::= "abc"+
```

and

```
<start> ::= "abc"{1,5}
```

respectively.

14.3.2 Regular Expressions over Bytes

Regular expressions can also be formed over bytes. See *Bytes and Regular Expressions* (page 104) for details.

14.4 Regular Expressions vs. Grammars

In theory, context-free grammars are a strict *superset* of regular expressions - any language that can be expressed in a regular expression can also be expressed in an equivalent grammar. Practical implementations of regular expressions break this hierarchy by introducing some features such as *backreferences* (check out what `(?P=name)` does), which cannot be expressed in grammars.

In many cases, a grammar can be replaced by a regular expression and vice versa. This raises the question: When should one use a regular expression, and when a grammar? Here are some points to help you decide.

- Regular expressions are often more *concise* (but arguably harder to read) than grammars.
- If you want to *reference* individual elements of a string (say, as part of a constraint now or in the future), use a *grammar*.
- Since their underlying model is simpler, regular expressions are *faster* to generate, and *much faster* to *parse* (page 97) than grammars.
- If your underlying language separates lexical and syntactical processing, use
  - *regular expressions* for specifying *lexical* parts such as tokens and fragments;
  - a *grammar* for the *syntax*; and
  - *constraints* (page 35) for any semantic properties.
- Prefer grammars and constraints over overly complex regular expressions.

Warning

Do not use regular expressions for inputs that are *recursive* (page 69). Languages like HTML, XML, even e-mail addresses or URLs, are much easier to capture as grammars.

14.5 Regular Expressions as Equivalence Classes

The choice of grammars vs. regular expressions also affects the Fandango generation algorithm. Generally speaking, Fandango attempts to cover all alternatives of a grammar. If, say, `<digits>` is specified as

```
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

then Fandango will attempt to produce every digit at least once, and also try to cover digit *combinations* up to a certain depth. This is useful if you want to specifically test digit processing, or if each of the digits causes a different behavior that needs to be covered.

If, however, you specify `<digits>` as

```
<digit> ::= r'[0-9]'
```

then Fandango will treat this as a *single* alternative (with all expansions considered semantically equivalent), which once expanded into (some) digit will be considered as covered.

Tip

- If you do want or need to *differentiate* between individual elements of a set (because they would be treated differently), consider *grammar alternatives*.
- If you do *not* want or need to differentiate between individual elements of a set (because they would all be treated the same), consider a *regular expression*.

CHAPTER

FIFTEEN

COMPLEX INPUT STRUCTURES

The *context-free grammars* that Fandango uses can specify very complex input formats. In particular, they allow specifying *recursive* inputs - that is, element types that can contain other elements of the same type again. In this chapter, we explore some typical patterns as they occur within grammars.

15.1 Recursive Inputs

A textbook example of a recursive input formats is an *arithmetic expression*. Consider an operation such as addition (+). Its operands can be *numbers* (`3 + 5`), but can also be *other expressions*, as in `2 + (3 - 8)`. In a grammar for arithmetic expressions, this relationship is expressed as a rule

Why don't we write `<expr> ::= <expr> " + " <expr>;` here? If we did that, a string such as `1 + 2 + 3` would become *ambiguous*: it will either be interpreted (1) as `(1 + 2) + 3` – that is, the left `<expr>` becomes `1 + 2`, and the right `<expr>` becomes `3`; or (2) as `1 + (2 + 3)` - that is, the left `<expr>` becomes `1`, and the right `<expr>` becomes `2 + 3`. Such ambiguities may not be important in *producing* inputs. But when *parsing* inputs, any ambiguities lead to interpretation and performance problems.

```
<expr> ::= <term> " + " <expr>
```

which indicates that the right-hand side of the addition + operator can be *another expression*. This is an example of a *recursive* grammar rule - a rule where an expansion refers back to the defined symbol.

Let us add a definition for `<term>`, too, defining it as a number:

```
<term> ::= <number>
<number>  ::= <digit>+
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

What is it that `<expr>` can now expand into? Have a look at the above rule first, and then check out the solution.

Solution

Actually, the above rules mean that `<expr>` would expand into an infinitely long string `<term> + <term> + <term> + ...`

What we have here is a case of *infinite recursion* – the string would keep on expanding forever.

Warning

At this time, Fandango does not detect infinite recursions; it keeps running until manually stopped.

In order to avoid infinite recursion, we need to provide a *non-recursive alternative*, as in:

```
<expr> ::= <term> " + " <expr> | <term>
```

With this rule, we can now store the above definitions in a `.fan` file additions.fan and get Fandango running:

```
$ fandango fuzz -f additions.fan -n 10
```

```
4
6
65
2 + 4
4960 + 8 + 74846 + 06 + 12
04011
691
547
07
21
```

We see that the above rules yield nice (recursive) chains of additions.

Warning

For each recursion in the grammar, there must be a non-recursive alternative.

15.2 More Repetitions

In our definition of `<number>`, above, we used the + operator to state that an element should be repeated:

```
<number> ::= <digit>+
```

Instead of using the + operator, though, we can also use a recursive rule. How would one do that?

We could also write

```
<number> ::= <number> <digit> | <digit>
```

However, if the recursive element is the *last* in a rule, this allows for more efficient parsing. We prefer such *tail recursion* whenever we can.

Solution

Here's an equivalent `<number>` definition that comes without `+`:

```
<number> ::= <digit> <number> | <digit>
```

Indeed, we can define a `<number>` as a `<digit>` that is followed by another “number”.

Hint

Using shorthands such as `*`, `+`, and `?` can make Fandango parsers more efficient.

15.3 Arithmetic Expressions

Let us put all of the above together into a grammar for arithmetic expressions. expr.fan defines additional operators (`-`, `*`, `/`), unary + and `-`, as well as subexpressions in parentheses. It also ensures the conventional order of operations[18], giving multiplication and division a higher rank than addition and subtraction. Hence, `2 * 3 + 4` gets interpreted as `(2 * 3) + 4` (10), not `2 * (3 + 4)` (14).

```
<start>   ::= <expr>
<expr>    ::= <term> " + " <expr> | <term> " - " <expr> | <term>
<term>    ::= <term> " * " <factor> | <term> " / " <factor> | <factor>
<factor>  ::= "+" <factor> | "-" <factor> | "(" <expr> ")" | <int>
<int>     ::= <digit>+
```

These are the expressions we get from this grammar:

```
$ fandango fuzz -f expr.fan -n 10
```

```
7
((7404 / 3 - 83)) / 73 * 4 - 89
(---3 + 82 * 68 * 94) * 89 / 92 / 23
+51 * (489 / 53 * 23) - 99
-+((7) * 34 / 66 / 24 * 5 * 21 - 00) - 8
((75 / 02 * 4 * 35 * 20 * 90 + 1) * 48 - 53) * 54
+84 * -(07 / 85 * 23 / 89 - 18) * 67 - 52
10 / 0 + (03) / 32 * 38 * 92 / 73 - 84
1187 * ++++++33 - 66
-1515 + +(49 + 7) + 12
```

We see that the resulting expressions can become quite complex. If we had an arithmetic evaluation to test – say, from a programming language - these would all make a good start.

Early Python versions interpreted zero-leading numbers as *octal* numbers, so `011` would evaluate to 9 (not 11). This was prone to errors, so in today’s Python, octal numbers need a `0o` prefix (as in `0o11`), and `0` prefixes yield an error.

There are two ways we can still improve the grammar, though.

1. First, many programming languages assign a special meaning to *numbers starting with zero*, so we’d like to get rid of these.

[18] https://en.wikipedia.org/wiki/Order_of_operations

2. Second, we only have *integers* so far. How would one go to extend the grammar with *floating-point numbers*?

Try this out for yourself by extending the above grammar.

Solution

To get rid of numbers starting with `0`, we can introduce a `<lead_digit>`:

```
<int> ::= <digit> | <lead_digit> <digits>
<lead_digit> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Note that the option of having a single `0` as an `<int>` is still there, as a `<digit>` can expand into any digit.

To include floating-point numbers, we can add a `<float>` element:

```
<float> ::= <int> "." <digits>
```

Feel free to further extend this - say, with an optional exponent, or by making the `<int>` optional (`.001`).

The resulting grammar `expr-float.fan` produces all-valid numbers:

```
$ fandango fuzz -f expr-float.fan -n 10
```

```
(3 * (-9 / 4 + 1) / 4 * 6) / 6 - 5
9 * (55554 / 1) / 1 - 1
+6.80 * 3 * -960.95 - 2
--+-602.25 * 6 / 6 / 8 * 2
(9 / (717.13 / 9) * 3) - 7
(--2.03 / 1 / 7 * 7 * 0 + 5) + 4
8775.5 + -6 / 3 * 9 / 3 - 8
(++(+7 * 9 - 3)) / 8 * 3 * 6 + 4
330 / +((2) * 8 / 4 + 6) / 4
259.452 + -468 / 3 - 3
```

With extra constraints, we can now have Fandango produce only expressions that satisfy further properties – say, evaluate to a value above 1000 in Python:

```
$ fandango fuzz -f expr-float.fan -n 10 -c 'eval(str(<start>)) > 1000'
```

```
8.99 / 8 * 50489
55198 * ((5 * 5))
```

```
(5449 + -4 - 0) - 4
+(933328) - 128.96 - 6
39638.1077 * 6 / 3 / 7 - 4
98050.672 / 9 * 1 / 3 * 2 - 5
1941
709747 - -1.0 * 3 * 9 - 1
(5 / 9 * 1 * 9 * 8 * 1 * 9 * 9 * 7 - 2) / 8 / 1
+68011 * 5.73 - 6
```

Note that some of these expressions raise divisions by zero errors:

```
ZeroDivisionError: float division by zero
```

In the next section, we'll talk about *accessing input elements* (page 73) in complex inputs, so we can impose further constraints on them.

CHAPTER

SIXTEEN

ACCESSING INPUT ELEMENTS

When dealing with *complex input formats* (page 69), attaching *constraints* (page 35) can be complex, as elements can occur *multiple times* in a generated input. Fandango offers a few mechanisms to disambiguate these, and to specify specific *contexts* in which to search for elements.

16.1 Derivation Trees

So far, we have always assumed that Fandango generates *strings* from the grammar. Behind the scenes, though, Fandango creates a far richer data structure - a so-called *derivation tree* that *maintains the structure of the string* and *allows accessing individual elements*. Every time Fandango sees a grammar rule

```
<symbol> ::= ...
```

it generates a derivation tree whose root is `<symbol>` and whose children are the elements of the right-hand side of the rule.

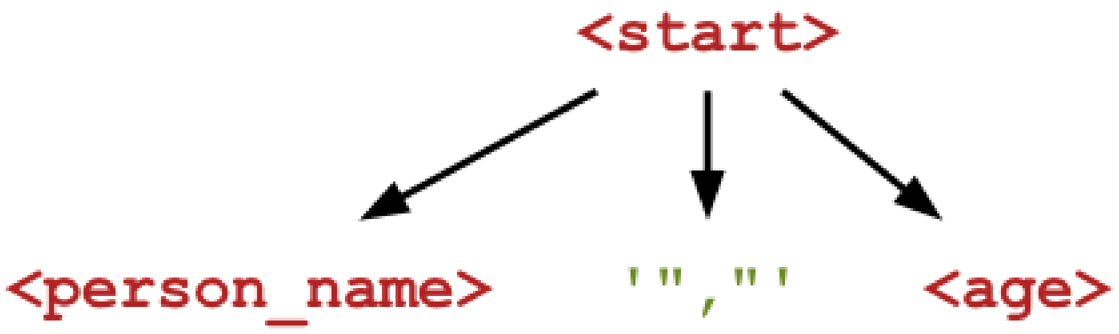
Let's have a look at our `persons.fan` spec:

```
<start> ::= <person_name> "," <age>
<person_name> ::= <first_name> " " <last_name>
<first_name> ::= <name>
<last_name> ::= <name>
<name> ::= <ascii_uppercase_letter><ascii_lowercase_letter>+
<age> ::= <digit>+
```

The `<start>` rule says

```
<start> ::= <person_name> "," <age>
```

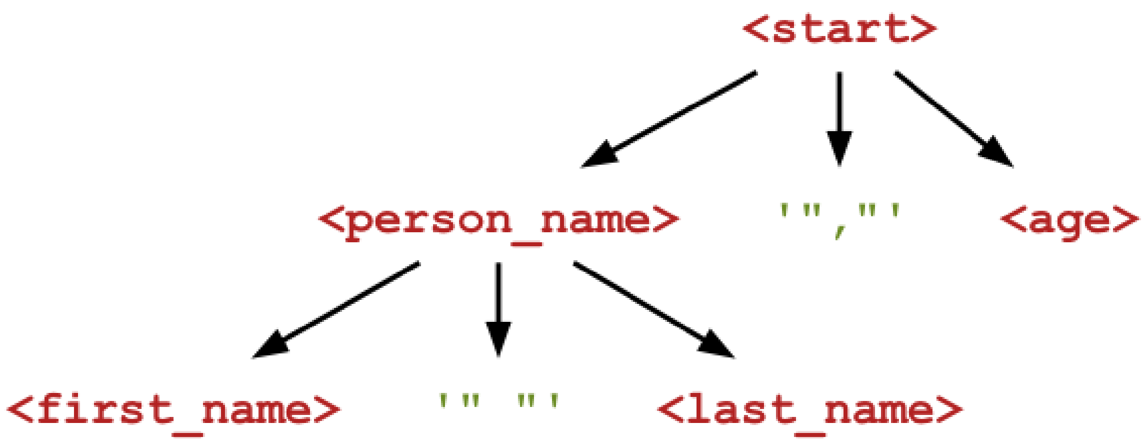
Then, a resulting derivation tree for `<start>` looks like this:



As Fandango expands more and more symbols, it expands the derivation tree accordingly. Since the grammar definition for `<person_name>` says

```
<person_name> ::= <first_name> " " <last_name>
```

the above derivation tree would be extended to

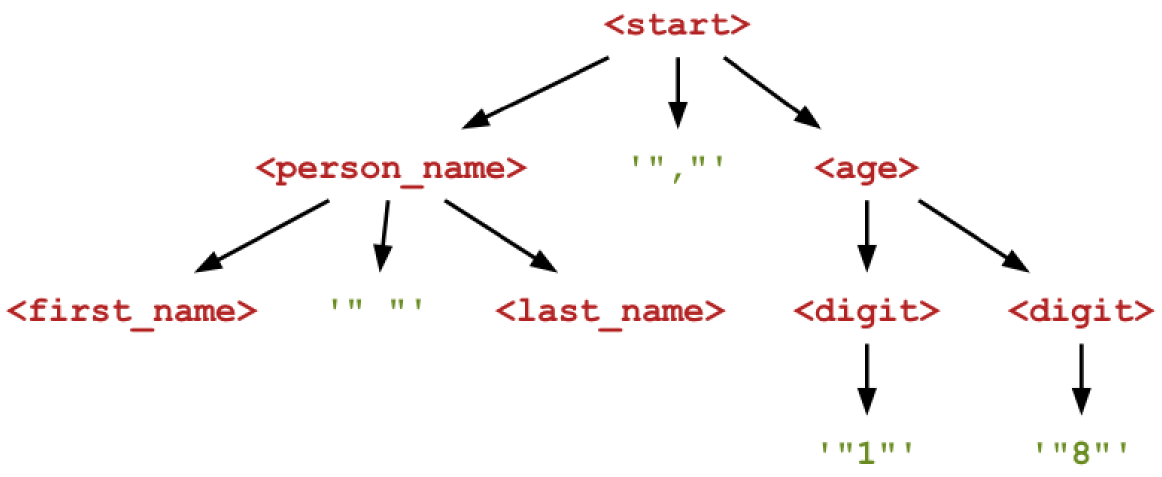


And if we next extend `<age>` and then `<digit>` based on their definitions

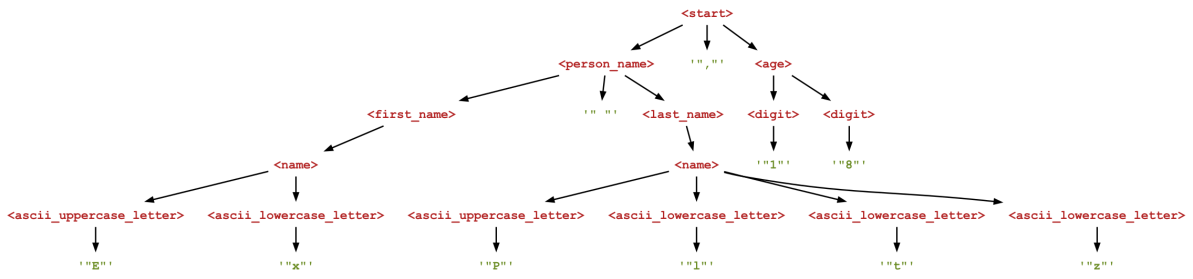
```
<age> ::= <digit>+
```

If one has a Kleene operator like `+`, `*`, or `?`, the elements operated on become children of the symbol being defined. Hence, the `<digit>` elements become children of the `<age>` symbol.

our tree gets to look like this:



Repeating the process, it thus obtains a tree like this:



Note how the tree records the entire history of how it was created - how it was *derived*, actually.

To obtain a string from the tree, we traverse its children left-to-right, ignoring all *nonterminal* symbols (in <...>) and considering only the *terminal* symbols (in quotes). This is what we get for the above tree:

```
Ex Pltz,18
```

And this is the string Fandango produces. However, viewing the Fandango results as derivation trees allows us to access *elements* of the Fandango-produced strings and to express *constraints* on them.

16.1.1 Diagnosing Derivation Trees

To examine the derivation trees that Fandango produces, use the *--format=grammar output format* (page 98). This produces the output in a grammar-like format, where children are indented under their respective parents. As an example, here is how to print a derivation tree from persons.fan:

```
$ fandango fuzz -f persons.fan -n 1 --format=grammar
```

```
<start> ::= <person_name> ',' <age>  # Position 0x0000 (0); 'Pl Seov,5150'
  <person_name> ::= <first_name> ' ' <last_name>  # Position 0x0001 (1); 'Pl Seov'
    <first_name> ::= <name>
      <name> ::= <ascii_uppercase_letter> <ascii_lowercase_letter>  # 'Pl'
        <ascii_uppercase_letter> ::= <_ascii_uppercase_letter>
          <_ascii_uppercase_letter> ::= 'P'  # Position 0x0002 (2)
        <ascii_lowercase_letter> ::= <_ascii_lowercase_letter>
          <_ascii_lowercase_letter> ::= 'l'  # Position 0x0003 (3)
```

(continues on next page)

(continued from previous page)

```
    <last_name> ::= <name>
      <name> ::= <ascii_uppercase_letter> <ascii_lowercase_letter> <ascii_
↪lowercase_letter> <ascii_lowercase_letter>  # 'Seov'
        <ascii_uppercase_letter> ::= <_ascii_uppercase_letter>
          <_ascii_uppercase_letter> ::= 'S'  # Position 0x0004 (4)
        <ascii_lowercase_letter> ::= <_ascii_lowercase_letter>
          <_ascii_lowercase_letter> ::= 'e'  # Position 0x0005 (5)
        <ascii_lowercase_letter> ::= <_ascii_lowercase_letter>
          <_ascii_lowercase_letter> ::= 'o'  # Position 0x0006 (6)
        <ascii_lowercase_letter> ::= <_ascii_lowercase_letter>
          <_ascii_lowercase_letter> ::= 'v'  # Position 0x0007 (7)
  <age> ::= <digit> <digit> <digit> <digit>  # '5150'
    <digit> ::= <_digit>
      <_digit> ::= '5'  # Position 0x0008 (8)
    <digit> ::= <_digit>
      <_digit> ::= '1'  # Position 0x0009 (9)
    <digit> ::= <_digit>
      <_digit> ::= '5'  # Position 0x000a (10)
    <digit> ::= <_digit>
      <_digit> ::= '0'  # Position 0x000b (11)
```

We see how the produced derivation tree consists of a `<start>` symbol, whose `<first_name>` and `<last_name>` children expand into `<name>` and letters; the `<age>` symbol expands into `<digit>` symbols.

The comments (after `#`) show the individual positions into the input, as well as the values of compound symbols.

What is the full string represented by the above derivation tree?

Solution

It's `'Pl Seov,5150'`, as you can find on the right-hand side of the first line.

Tip

The `--format=grammar` option is great for debugging, especially *binary formats* (page 101).

16.2 Specifying Paths

One effect of Fandango producing derivation trees rather than "just" strings is that we can define special *operators* that allow us to access *subtrees* (or sub-elements) of the produced strings - and express constraints on them. This is especially useful if we want constraints to apply only in specific *contexts* - say, as part of some element `<a>`, but not as part of an element `<b>`.

16.2.1 Accessing Children

These selectors are similar to XPath, but better aligned with Python. In XPath, the first child has the index 1, in Fandango, it has the index 0.

The expression `<foo>[N]` accesses the `N`-th child of `<foo>`, starting with zero.

If `<foo>` is defined in the grammar as

```
<foo> ::= <bar> ":" <baz>
```

then `<foo>[0]` returns the `<bar>` element, `<foo>[1]` returns `":"`, and `<foo>[2]` returns the `<baz>` element.

In our *`persons.fan` derivation tree for `Ex Pltz`* (page 73), for instance, `<start>[0]` would return the `<person_name>` element (`"Ex Pltz"`), and `<start>[2]` would return the `<age>` element (`18`).

We can use this to access elements in specific contexts. For instance, if we want to refer to a `<name>` element, but only if it is the child of a `<first_name>` element, we can refer to it as `<first_name>[0]` - the first child of a `<first_name>` element:

```
<first_name> ::= <name>
```

Here is a constraint that makes Fandango produce first names that end with `x`:

Since a `<first_name>` is defined to be a `<name>`, we could also write `<first_name>.endswith("x")`

```
$ fandango fuzz -f persons.fan -n 10 -c '<first_name>[0].endswith("x")'
```

```
Gux Wmt,233
Keeqtx Ueps,60
Ax Wz,7
Fex Yhi,02845
Wlytdx Wptfbd,38
Xx Xok,17
Fex Srorbr,4
Lix Zll,5
Gux Ueps,233
Keeqtx Wmt,60
```

Note

As in Python, you can use *negative* indexes to refer to the last elements. `<age>[-1]`, for instance, gives you the *last* child of an `<age>` subtree.

Warning

While symbols act as strings in many contexts, this is where they differ. To access the first *character* of a symbol `<foo>`, you need to explicitly convert it to a string first, as in `str(<foo>)[0]`.

16.2.2 Slices

The Fandango slices maintain the property that `<name>[n:m]` = `<name>[n]` + `<name>[n + 1]` + … + `<name>[m - 1]`, which holds for all sequences in Python.

Fandango also allows you to use Python *slice* syntax to access *multiple children at once*. `<name>[n:m]` returns a new (unnamed) root which has `<name>[n]`, `<name>[n + 1]`, …, `<name>[m - 1]` as children. This is useful, for instance, if you want to compare several children against a string:

```
$ fandango fuzz -f persons-faker.fan -n 10 -c '<name>[0:2] == "Ch"'
```

```
Chelsea Chung,685
Christian Chung,9
Christopher Chung,685
Christopher Chung,75
Christopher Chan,00
Chelsea Chung,75
Christopher Chan,7
Chelsea Chung,194
Christopher Chavez,00
Christopher Chung,0
```

Would one also be able to use `<start>[0:2] == "Ch"` to obtain inputs that all start with `"Ch"`?

Solution

No, this would not work. Remember that in derivation trees, indexes refer to *children*, not characters. So, according to the rule

```
<start> ::= <person_name> "," <age>
```

`<start>[0]` is a `<person_name>`, `<start>[1]` is a `","`, and `<start>[2]` is an `<age>`. Hence, `<start>[0:2]` refers to `<start>` itself, which cannot be `"Ch"`.

Indeed, to have the *string* start with `"Ch"`, you (again) need to convert `<start>` into a string first, and then access its individual characters:

```
$ fandango fuzz -f persons-faker.fan -n 10 -c 'str(<start>)[0:2] == "Ch"'
```

```
Christina Frederick,38
Christine Mckenzie,48
Christopher Hall,50
Chad Richardson,8
Christopher Ramirez,54
Christopher Edwards,09
```

(continues on next page)

(continued from previous page)

```
Christopher Kelly,76
Christian Wilson,10
Christopher Beltran,09
Chase Calhoun,26
```

The Fandango slices maintain the property that `<name>[i:]` + `<name>[:i]` = `<name>`

Fandango supports the full Python slice semantics:

- An omitted first index defaults to zero, so `<foo>[:2]` returns the first two children.
- An omitted second index defaults to the size of the string being sliced, so `<foo>[2:]` returns all children starting with `<foo>[2]`.
- Both the first and the second index can be negative again.

16.2.3 Selecting Children

Referring to children by *number*, as in `<foo>[0]`, can be a bit cumbersome. This is why in Fandango, you can also refer to elements by *name*.

In XPath, the corresponding operator is `/`.

The expression `<foo>.<bar>` allows accessing elements `<bar>` when they are a *direct child* of a symbol `<foo>`. This requires that `<bar>` occurs in the grammar rule defining `<foo>`:

```
<foo> ::= ...some expansion that has <bar>...
```

To refer to the `<name>` element as a direct child of a `<first_name>` element, you thus write `<first_name>.<name>`. This allows you to express the earlier constraint in a possibly more readable form:

```
$ fandango fuzz -f persons.fan -n 10 -c '<first_name>.<name>.endswith("x")'
```

```
Tx Gx,1
Ex Sbkeg,8425
Wwxx Nrtgs,96
Truehx Htbiw,84
Vex Kzqok,7
Vekx Iu,195
Truexx Htbiw,84
Kufxx Fbblqf,6
Dwx Zatr,63
Ebvx Sqife,9333
```

Note

You can only access *nonterminal* children this way; `<person_name>." "` (the space in the `<person_name>`) gives an error.

16.2.4 Selecting Descendants

Often, you want to refer to elements in a particular *context* set by the enclosing element. This is why in Fandango, you can also refer to *descendants*.

In XPath, the corresponding operator is `//`.

The expression `<foo>..<bar>` allows accessing elements `<bar>` when they are a *descendant* of a symbol `<foo>`. `<bar>` is a descendant of `<foo>` if

`<foo>..<bar>` includes `<foo>.<bar>`.

- `<bar>` is a child of `<foo>`; or
- one of `<foo>`'s children has `<bar>` as a descendant.

If that sounds like a recursive definition, that is because it is. A simpler way to think about `<foo>..<bar>` may be "All `<bar>`s that occur within `<foo>`".

Let us take a look at some rules in our `persons.fan` example:

```
<first_name> ::= <name>
<last_name> ::= <name>
<name> ::= <ascii_uppercase_letter><ascii_lowercase_letter>+
<ascii_uppercase_letter> ::= "A" | "B" | "C" | ... | "Z"
```

To refer to all `<ascii_uppercase_letter>` element as descendant of a `<first_name>` element, you thus write `<first_name>..<ascii_uppercase_letter>`.

Hence, to make all uppercase letters `X`, but only as part of a first name, you may write

```
$ fandango fuzz -f persons.fan -n 10 -c '<first_name>..<ascii_uppercase_letter> == "X"
↪'
```

```
Xr Clnuvu,610
Xt Uggf,63928
Xkbsxy Kt,61140
Xuxeh Awa,97
Xew Wgxidg,3708
Xga Gbruqf,6
Xz Pmqs,11160
Xsitcd Pff,61
```

(continues on next page)

(continued from previous page)

```
Xl Azkva,0
Xb Tjkx,119
```

16.2.5 Chains

You can freely combine `[]`, `.`, and `..` into *chains* that select subtrees. What would the expression `<start>[0].<last_name>..<ascii_lowercase_letter>` refer to, for instance?

Solution

This is easy:

- `<start>[0]` is the first element of start, hence a `<person_name>`.
- `.<last_name>` refers to the child of type `<last_name>`
- `..<ascii_lowercase_letter>` refers to all descendants of type `<ascii_lowercase_letter>`

Let's use this in a constraint:

```
$ fandango fuzz -f persons.fan -n 10 -c '<start>[0].<last_name>..<ascii_lowercase_
↪letter> == "x"'
```

```
Yqxxum Jxx,7392
Nusga Kxx,997
Jmeqvm Bx,25817
Nqcmx Kxxx,001
Fx Oxxxxx,05
Kmfmk Kxxxx,7
Qejux Axxxx,888
Dgiiv Oxxxxx,4
Mcgy Gxxx,47083
Ngrlpx Jx,883
```

16.3 Quantifiers

By default, whenever you use a symbol `<foo>` in a constraint, this constraint applies to *all* occurrences of `<foo>` in the produced output string. For your purposes, though, one such instance already may suffice. This is why Fandango allows expressing *quantification* in constraints.

16.3.1 Star Expressions

Error

The `*` syntax is not operational yet.

In Fandango, you can prefix an element with `*` to obtain a collection of *all* these elements within an individual string. Hence, `*<name>` is a collection of *all* `<name>` elements within the generated string. This syntax can be used in a variety of ways. For instance, we can have a constraint check whether a particular element is in the collection:

Not every constraint that can be *expressed* also can be *solved* by Fandango.

```
"Pablo" in *<name>
```

This constraint evaluates to True if any of the values in `*<name>` (= one of the two `<name>` elements) is equal to `"Pablo"`. `*`-expressions are mostly useful in quantifiers, which we discuss below.

16.3.2 Existential Quantification

To express that within a particular scope, at least one instance of a symbol must satisfy a constraint, use a `*`-expression in combination with `any()`:

```
any(CONSTRAINT for ELEM in *SCOPE)
```

where

- `SCOPE` is a nonterminal (e.g. `<age>`);
- `ELEM` is a Python variable; and
- `CONSTRAINT` is a constraint over `ELEM`

Hence, the expression

```
any(n.startswith("A") for n in *<name>)
```

ensures there is at least *one* element `<name>` that starts with an “A”:

Let us decompose this expression for a moment:

- The expression `for n in *<name>` lets Python iterate over `*<name>` (all `<name>` objects within a person)…
- … and evaluate `n.startswith("A")` for each of them, resulting in a collection of Boolean values.
- The Python function `any(list)` returns `True` if at least one element in `list` is True.

So, what we get is existential quantification:

```
$ fandango fuzz -f persons.fan -n 10 -c 'any(n.startswith("A") for n in *<name>)'
```

```
Akj Obsfz,536
Ac Wrbe,687
Ajlc Dejnph,582
Akoht Hgx,63
Athtq Kap,48
Akz Wexswj,474
Afcn Adxn,532
Amd Devz,42254
Ajlc Dejnph,482
Ahddx Emjgw,3
```

16.3.3 Universal Quantification

Where there are existential quantifiers, there also are *universal* quantifiers. Here we use the Python `all()` function; `all(list)` evaluates to True only if *all* elements in `list` are True.

We use a `*`-expression in combination with `all()`:

```
all(CONSTRAINT for ELEM in *SCOPE)
```

Hence, the expression

```
all(c == "a" for c in *<first_name>..<ascii_lowercase_letter>)
```

ensures that *all* sub-elements `<ascii_lowercase_letter>` in `<first_name>` have the value "a".

Again, let us decompose this expression:

- The expression `for c in *<first_name>..<ascii_lowercase_letter>` lets Python iterate over all `<ascii_lowercase_letter>` objects within `<first_name>`…
- … and evaluate `c == "a"` for each of them, resulting in a collection of Boolean values.
- The Python function `all(list)` returns `True` if all elements in `list` are True.

So, what we get is universal quantification:

```
$ fandango fuzz -f persons.fan -n 10 -c 'all(c == "a" for c in *<first_name>..<ascii_
↪lowercase_letter>)'
```

```
Ea Woeen,95212
Aaaaaa Ihrplw,47
Waaaaa Ieba,624
Naaaaa Vdnj,5
Uaaaa Pwe,66455
Raa Zacjhe,9
Fa Yapa,993
Naaaa Xzru,50
Caaaaa Dgaqht,4
Haaaa Mthga,967
```

By default, all symbols are universally quantified within `<start>`, so a dedicated universal quantifier is only needed if you want to *limit the scope* to some sub-element. This is what we do here with `<first_name>..<ascii_lowercase_letter>`, limiting the scope to `<first_name>`.

Given the default universal quantification, you can actually achieve the same effect as above without `all()` and without a `*`. How?

Solution

You can access all `<ascii_lowercase_letter>` elements within `<first_name>` directly:

```
$ fandango fuzz -f persons.fan -n 10 -c '<first_name>..<ascii_lowercase_letter> ==
↪"a"'
```

CHAPTER

SEVENTEEN

CASE STUDY: ISO 8601 DATE + TIME

Let us make use of what we have explored so far in a more elaborate case study. ISO 8601[19] is an international standard for exchange and communication of *date and time-related data*, providing an unambiguous method of representing calendar dates and times.

In this chapter, we will define a full Fandango spec for ISO 8601 date and time formats, ensuring both syntactic and semantic validity. To make matters more interesting, we will create the spec *programmatically* - that is, by having a number of Python functions that generate the spec for us.

17.1 Creating Grammars Programmatically

We start with a number of functions that help us create fragments of the grammar.

`make_rule()` creates a grammar rule from `symbol` and its possible `expansions`:

```python
import sys

def print_s(s: str) -> str:
    print(s, end="")
    return s

def make_rule(symbol: str, expansions: list[str], sep: str = '') -> str:
    return print_s(f"{sep}<{symbol}> ::= " + " | ".join(expansions) + "\n")

make_rule("start", ["<iso8601datetime>"]);  # a final ";" suppresses result
```

```
<start> ::= <iso8601datetime>
```

We can also use `make_rule()` to quickly create a list of expansions:

```python
make_rule("digit", [f"'{digit}'" for digit in range(0, 10)]);
```

```
<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

These additional functions help in adding non-grammar elements such as headers, code, and constraints:

```python
def make_header(title: str) -> str:
    return print_s(f"\n# {title}\n")

def make_comment(comment: str, sep: str = '') -> str:
```

(continues on next page)

[19] https://en.wikipedia.org/wiki/ISO_8601

(continued from previous page)

```
    return print_s(f"{sep}# {comment}\n")

def make_constraint(constraint: str) -> str:
    return print_s(f"where {constraint}\n")

def make_code(code: str) -> str:
    return print_s(f"{code}\n")

make_header("ISO 8601 grammar");
```

```
# ISO 8601 grammar
```

17.2 Spec Header

Let us create a Fandango spec. For now, we store the final spec in the `iso8601lib` string variable, to be saved in a file at the end. Every time we add a new fragment, the new fragment will be output here, so we can see the actual content of `iso8601lib` incrementally.

```
iso8601lib = make_header("ISO 8601 date and time")
iso8601lib += make_comment("Generated by docs/ISO8601.md. Do not edit.")
```

```
# ISO 8601 date and time
# Generated by docs/ISO8601.md. Do not edit.
```

We will need the `datetime` library below, so we import it.

```
iso8601lib += make_code("\nimport datetime\n")
```

```
import datetime
```

17.3 Date

We start with a `<start>` symbol. An ISO 8601 date/time spec starts with a date, and an optional time, separated by `T`:

```
iso8601lib += make_rule("start", ["<iso8601datetime>"])

iso8601lib += make_rule("iso8601datetime",
                    ["<iso8601date> ('T' <iso8601time>)?"])
```

```
<start> ::= <iso8601datetime>
<iso8601datetime> ::= <iso8601date> ('T' <iso8601time>)?
```

A date can either be a *calendar date*, but also a *week date* or an *ordinal date*.

```
iso8601lib += make_rule("iso8601date",
                        ["<iso8601calendardate>",
                         "<iso8601weekdate>",
                         "<iso8601ordinaldate>"], sep='\n')
```

```
<iso8601date> ::= <iso8601calendardate> | <iso8601weekdate> | <iso8601ordinaldate>
```

17.3.1 Calendar Dates

An ISO 8601 calendar date has the format YYYY-MM-DD, but can also be YYYY-MM or YYYYMMDD. The year can be prefixed with + or − to indicate AC/BC[20] (or CE/BCE[21]) designators.

```
iso8601lib += make_rule("iso8601calendardate",
                        ["<iso8601year> '-' <iso8601month> ('-' <iso8601day>)?",
                         "<iso8601year> <iso8601month> <iso8601day>"], sep='\n')
iso8601lib += make_rule("iso8601year", ["('+'|'-')? <digit>{4}"])
```

```
<iso8601calendardate> ::= <iso8601year> '-' <iso8601month> ('-' <iso8601day>)? |
↪<iso8601year> <iso8601month> <iso8601day>
<iso8601year> ::= ('+'|'-')? <digit>{4}
```

17.3.2 Months

Months are easy:

```
iso8601lib += make_rule("iso8601month",
                        [f"'{month:02d}'" for month in range(1, 13)])
```

```
<iso8601month> ::= '01' | '02' | '03' | '04' | '05' | '06' | '07' | '08' | '09' |
↪'10' | '11' | '12'
```

17.3.3 Days

As with months, we define each day individually to avoid biasing the grammar. The fact that some months have only 28, 29, or 30 days will be handled later on in a constraint.

```
iso8601lib += make_rule("iso8601day",
                        [f"'{day:02d}'" for day in range(1, 32)])
```

```
<iso8601day> ::= '01' | '02' | '03' | '04' | '05' | '06' | '07' | '08' | '09' | '10
↪' | '11' | '12' | '13' | '14' | '15' | '16' | '17' | '18' | '19' | '20' | '21' |
↪'22' | '23' | '24' | '25' | '26' | '27' | '28' | '29' | '30' | '31'
```

Tip

For testing purposes, it makes sense to test with *extreme* values - in our case, January 1, February 28 or 29, and December 31.

[20] https://en.wikipedia.org/wiki/Anno_Domini
[21] https://en.wikipedia.org/wiki/Common_Era

17.3.4 Week Dates

ISO 8601 also allows specifying dates by week numbers (1 - 53) and optional days of the week (1 - 7). `2025-W12-1` is the Monday of the 12th week in 2025.

```
iso8601lib += make_rule("iso8601weekdate",
                        ["<iso8601year> '-'? 'W' <iso8601week> ('-' <iso8601weekday>)?"],␣
↪sep='\n')
iso8601lib += make_rule("iso8601week",
                        [f"'{week:02d}'" for week in range(1, 54)])
iso8601lib += make_rule("iso8601weekday",
                        [f"'{weekday:1d}'" for weekday in range(1, 8)])
```

```
<iso8601weekdate> ::= <iso8601year> '-'? 'W' <iso8601week> ('-' <iso8601weekday>)?
<iso8601week> ::= '01' | '02' | '03' | '04' | '05' | '06' | '07' | '08' | '09' |
↪'10' | '11' | '12' | '13' | '14' | '15' | '16' | '17' | '18' | '19' | '20' | '21
↪' | '22' | '23' | '24' | '25' | '26' | '27' | '28' | '29' | '30' | '31' | '32' |
↪'33' | '34' | '35' | '36' | '37' | '38' | '39' | '40' | '41' | '42' | '43' | '44
↪' | '45' | '46' | '47' | '48' | '49' | '50' | '51' | '52' | '53'
<iso8601weekday> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7'
```

17.3.5 Ordinal Dates

ISO 8601 also allows specifying dates by day numbers (1 - 366). `2000-366`, for instance, specifies the last day in 2000 (which was a leap year). Again, we specify each day programmatically (and individually).

```
iso8601lib += make_rule("iso8601ordinaldate",
                        ["<iso8601year> ('-'? <iso8601ordinalday>)?"], sep='\n')
iso8601lib += make_rule("iso8601ordinalday",
                        [f"'{day:03d}'" for day in range(1, 367)])
```

```
<iso8601ordinaldate> ::= <iso8601year> ('-'? <iso8601ordinalday>)?
<iso8601ordinalday> ::= '001' | '002' | '003' | '004' | '005' | '006' | '007' |
↪'008' | '009' | '010' | '011' | '012' | '013' | '014' | '015' | '016' | '017' |
↪'018' | '019' | '020' | '021' | '022' | '023' | '024' | '025' | '026' | '027' |
↪'028' | '029' | '030' | '031' | '032' | '033' | '034' | '035' | '036' | '037' |
↪'038' | '039' | '040' | '041' | '042' | '043' | '044' | '045' | '046' | '047' |
↪'048' | '049' | '050' | '051' | '052' | '053' | '054' | '055' | '056' | '057' |
↪'058' | '059' | '060' | '061' | '062' | '063' | '064' | '065' | '066' | '067' |
↪'068' | '069' | '070' | '071' | '072' | '073' | '074' | '075' | '076' | '077' |
↪'078' | '079' | '080' | '081' | '082' | '083' | '084' | '085' | '086' | '087' |
↪'088' | '089' | '090' | '091' | '092' | '093' | '094' | '095' | '096' | '097' |
↪'098' | '099' | '100' | '101' | '102' | '103' | '104' | '105' | '106' | '107' |
↪'108' | '109' | '110' | '111' | '112' | '113' | '114' | '115' | '116' | '117' |
↪'118' | '119' | '120' | '121' | '122' | '123' | '124' | '125' | '126' | '127' |
↪'128' | '129' | '130' | '131' | '132' | '133' | '134' | '135' | '136' | '137' |
↪'138' | '139' | '140' | '141' | '142' | '143' | '144' | '145' | '146' | '147' |
↪'148' | '149' | '150' | '151' | '152' | '153' | '154' | '155' | '156' | '157' |
↪'158' | '159' | '160' | '161' | '162' | '163' | '164' | '165' | '166' | '167' |
↪'168' | '169' | '170' | '171' | '172' | '173' | '174' | '175' | '176' | '177' |
↪'178' | '179' | '180' | '181' | '182' | '183' | '184' | '185' | '186' | '187' |
↪'188' | '189' | '190' | '191' | '192' | '193' | '194' | '195' | '196' | '197' |
↪'198' | '199' | '200' | '201' | '202' | '203' | '204' | '205' | '206' | '207' |
↪'208' | '209' | '210' | '211' | '212' | '213' | '214' | '215' | '216' | '217' |
↪'218' | '219' | '220' | '221' | '222' | '223' | '224' | '225' | '226' | '227' |
```

(continues on next page)

(continued from previous page)

```
↪'228' | '229' | '230' | '231' | '232' | '233' | '234' | '235' | '236' | '237' |
↪'238' | '239' | '240' | '241' | '242' | '243' | '244' | '245' | '246' | '247' |
↪'248' | '249' | '250' | '251' | '252' | '253' | '254' | '255' | '256' | '257' |
↪'258' | '259' | '260' | '261' | '262' | '263' | '264' | '265' | '266' | '267' |
↪'268' | '269' | '270' | '271' | '272' | '273' | '274' | '275' | '276' | '277' |
↪'278' | '279' | '280' | '281' | '282' | '283' | '284' | '285' | '286' | '287' |
↪'288' | '289' | '290' | '291' | '292' | '293' | '294' | '295' | '296' | '297' |
↪'298' | '299' | '300' | '301' | '302' | '303' | '304' | '305' | '306' | '307' |
↪'308' | '309' | '310' | '311' | '312' | '313' | '314' | '315' | '316' | '317' |
↪'318' | '319' | '320' | '321' | '322' | '323' | '324' | '325' | '326' | '327' |
↪'328' | '329' | '330' | '331' | '332' | '333' | '334' | '335' | '336' | '337' |
↪'338' | '339' | '340' | '341' | '342' | '343' | '344' | '345' | '346' | '347' |
↪'348' | '349' | '350' | '351' | '352' | '353' | '354' | '355' | '356' | '357' |
↪'358' | '359' | '360' | '361' | '362' | '363' | '364' | '365' | '366'
```

Tip

For testing purposes, it makes sense to test with *extreme* values - in our case, `1`, `365` and `366`.

17.4 Time

Now for *time* specifications. In ISO 8601, time is specified as `HH:MM:SS`, where `HH` comes in 24-hour format. `MM` and `SS` are optional, as is the colon separator in an “unambiguous” context.

```
iso8601lib += make_rule("iso8601time",
                        ["'T'? <iso8601hour> (':'? <iso8601minute> (':'? <iso8601second>␣
↪(('.' | ',') <iso8601fraction>)? )? )? <iso8601timezone>?"], sep='\n')
```

```
<iso8601time> ::= 'T'? <iso8601hour> (':'? <iso8601minute> (':'? <iso8601second> ((
↪'.' | ',') <iso8601fraction>)? )? )? <iso8601timezone>?
```

17.4.1 Hours

Hours normally go from `00` to `23`, but `24:00:00` is allowed to represent midnight at the end of a day.

```
iso8601lib += make_comment("24:00:00 is allowed to represent midnight at the end of a␣
↪day", sep='\n')
iso8601lib += make_rule("iso8601hour",
                        [f"'{hour:02d}'" for hour in range(0, 25)])
```

```
# 24:00:00 is allowed to represent midnight at the end of a day
<iso8601hour> ::= '00' | '01' | '02' | '03' | '04' | '05' | '06' | '07' | '08' |
↪'09' | '10' | '11' | '12' | '13' | '14' | '15' | '16' | '17' | '18' | '19' | '20
↪' | '21' | '22' | '23' | '24'
```

17.4.2 Minutes

Minutes go from `00` to `59`. Nothing special here.

```
iso8601lib += make_rule("iso8601minute",
                        [f"'{minute:02d}'" for minute in range(0, 60)])
```

```
<iso8601minute> ::= '00' | '01' | '02' | '03' | '04' | '05' | '06' | '07' | '08' |
↪'09' | '10' | '11' | '12' | '13' | '14' | '15' | '16' | '17' | '18' | '19' | '20
↪' | '21' | '22' | '23' | '24' | '25' | '26' | '27' | '28' | '29' | '30' | '31' |
↪'32' | '33' | '34' | '35' | '36' | '37' | '38' | '39' | '40' | '41' | '42' | '43
↪' | '44' | '45' | '46' | '47' | '48' | '49' | '50' | '51' | '52' | '53' | '54' |
↪'55' | '56' | '57' | '58' | '59'
```

17.4.3 Seconds

Seconds normally go from `00` to `59`, but `60` can be used to represent leap seconds.

```
iso8601lib += make_comment("xx:yy:60 is allowed to represent leap seconds", sep='\n')
iso8601lib += make_rule("iso8601second",
                        [f"'{second:02d}'" for second in range(0, 61)])
```

```
# xx:yy:60 is allowed to represent leap seconds
<iso8601second> ::= '00' | '01' | '02' | '03' | '04' | '05' | '06' | '07' | '08' |
↪'09' | '10' | '11' | '12' | '13' | '14' | '15' | '16' | '17' | '18' | '19' | '20
↪' | '21' | '22' | '23' | '24' | '25' | '26' | '27' | '28' | '29' | '30' | '31' |
↪'32' | '33' | '34' | '35' | '36' | '37' | '38' | '39' | '40' | '41' | '42' | '43
↪' | '44' | '45' | '46' | '47' | '48' | '49' | '50' | '51' | '52' | '53' | '54' |
↪'55' | '56' | '57' | '58' | '59' | '60'
```

Seconds can be followed by `,` or a `.` (see `<iso8601time>`, above) and a fraction - an arbitrary number of digits.

```
iso8601lib += make_rule("iso8601fraction", ["<digit>+"])
```

```
<iso8601fraction> ::= <digit>+
```

Tip

Testing with *extreme* values would mandate times such as `20:15:00,99999999999999999999` to test for possible buffer overflows.

17.4.4 Time Zones

Any time can be followed by a time zone designator. This is either `Z`, indicating Coordinated Universal Time (UTC), or an offset designating the time zone.

```
iso8601lib += make_rule("iso8601timezone", ["'Z'",
                                            "'+' <iso8601hour> (':'? <iso8601minute>)?
↪",
                                            "'-' <iso8601hour> (':'? <iso8601minute>)?
↪"], sep='\n')
```

```
<iso8601timezone> ::= 'Z' | '+' <iso8601hour> (':'? <iso8601minute>)? | '-'
↪<iso8601hour> (':'? <iso8601minute>)?
```

17.5 Ensuring Validity

So far, our dates and times are *syntactically* valid, but not necessarily *semantically*. We can easily create invalid dates such as `2025-02-31` (February 31) or invalid times such as `24:10:00` (10 minutes past midnight - only `24:00:00` is allowed). We could now add a number of additional *rules* to check for all these properties, and/or extend the grammar accordingly. However, we can also simply be lazy and have the existing Python `datetime` module check all this for us:

```python
import datetime
```

```python
def is_valid_iso8601datetime(iso8601datetime: str) -> bool:
    """Return True iff `iso8601datetime` is valid."""
    try:
        datetime.datetime.fromisoformat(iso8601datetime)
        return True
    except ValueError:
        return False
```

```python
is_valid_iso8601datetime("2025-01-01T14:00")
```

```
True
```

```python
is_valid_iso8601datetime("2025-02-31")
```

```
False
```

```python
is_valid_iso8601datetime("2000-02-29T00:00:00")
```

```
True
```

With this, we can now add a *constraint* that limits our generator to only valid dates and times:

```python
iso8601lib += make_constraint("is_valid_iso8601datetime(str(<iso8601datetime>))")
```

```
where is_valid_iso8601datetime(str(<iso8601datetime>))
```

```python
iso8601lib += '''
def is_valid_iso8601datetime(iso8601datetime: str) -> bool:
    """Return True iff `iso8601datetime` is valid."""
    try:
        datetime.datetime.fromisoformat(iso8601datetime)
        return True
    except ValueError:
        return False
'''
```

17.6 Even Better Validity

The Python `datetime` module has a some limitations, which extend to our spec. For instance, `datetime` does not support `24:00:00` as a valid time:

```
is_valid_iso8601datetime("2024-12-31T24:00:00")
```

```
False
```

Hence, our `iso8601.fan` spec may miss out a number of ISO 8601 features.

The `dateutil` alternative[22] provides an ISO 8601 parser without these deficiencies.

Let us redefine `is_valid_iso8601datetime()` to make use of the `dateutil` parser:

```
def is_valid_iso8601datetime(iso8601datetime: str) -> bool:
    """Return True iff `iso8601datetime` is valid."""
    try:
        dateutil.parser.isoparse(iso8601datetime)
        return True
    except ValueError:
        return False
```

Let us see if the above example now works:

```
is_valid_iso8601datetime("2024-12-31T24:00:00")
```

```
True
```

This now works! How about the earlier examples?

```
is_valid_iso8601datetime("2025-01-01T14:00")
```

```
True
```

```
is_valid_iso8601datetime("2025-02-31")
```

```
False
```

```
is_valid_iso8601datetime("2000-02-29T00:00:00")
```

```
True
```

These also work. Let us fix the spec to use `dateutil` instead:

```
iso8601lib = iso8601lib.replace('datetime.datetime.fromisoformat', 'dateutil.parser.
↪isoparse')
iso8601lib = iso8601lib.replace('import datetime', 'import dateutil  # See https://
↪dateutil.readthedocs.io');
```

[22] https://dateutil.readthedocs.io/en/stable/index.html

17.7 Fuzzing Dates and Times

Our ISO 8601 spec is now complete. Let us write it into a `.fan` file, so we can use it for fuzzing:

```
open('ISO8601.fan', 'w').write(iso8601lib);
```

Here comes `iso8601.fan` in all its glory:

```
# ISO 8601 date and time
# Generated by docs/ISO8601.md. Do not edit.

import dateutil  # See https://dateutil.readthedocs.io

<start> ::= <iso8601datetime>
<iso8601datetime> ::= <iso8601date> ('T' <iso8601time>)?

<iso8601date> ::= <iso8601calendardate> | <iso8601weekdate> |
<iso8601ordinaldate>

<iso8601calendardate> ::= <iso8601year> '-' <iso8601month> ('-' <iso8601day>)?
| <iso8601year> <iso8601month> <iso8601day>
<iso8601year> ::= ('+'|'-')? <digit>{4}
<iso8601month> ::= '01' | '02' | '03' | '04' | '05' | '06' | '07' | '08' | '09'
| '10' | '11' | '12'
<iso8601day> ::= '01' | '02' | '03' | '04' | '05' | '06' | '07' | '08' | '09' |
'10' | '11' | '12' | '13' | '14' | '15' | '16' | '17' | '18' | '19' | '20' |
'21' | '22' | '23' | '24' | '25' | '26' | '27' | '28' | '29' | '30' | '31'

<iso8601weekdate> ::= <iso8601year> '-'? 'W' <iso8601week> ('-'
<iso8601weekday>)?
<iso8601week> ::= '01' | '02' | '03' | '04' | '05' | '06' | '07' | '08' | '09'
| '10' | '11' | '12' | '13' | '14' | '15' | '16' | '17' | '18' | '19' | '20' |
'21' | '22' | '23' | '24' | '25' | '26' | '27' | '28' | '29' | '30' | '31' |
'32' | '33' | '34' | '35' | '36' | '37' | '38' | '39' | '40' | '41' | '42' |
'43' | '44' | '45' | '46' | '47' | '48' | '49' | '50' | '51' | '52' | '53'
<iso8601weekday> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7'

<iso8601ordinaldate> ::= <iso8601year> ('-'? <iso8601ordinalday>)?
<iso8601ordinalday> ::= '001' | '002' | '003' | '004' | '005' | '006' | '007' |
'008' | '009' | '010' | '011' | '012' | '013' | '014' | '015' | '016' | '017' |
'018' | '019' | '020' | '021' | '022' | '023' | '024' | '025' | '026' | '027' |
'028' | '029' | '030' | '031' | '032' | '033' | '034' | '035' | '036' | '037' |
'038' | '039' | '040' | '041' | '042' | '043' | '044' | '045' | '046' | '047' |
'048' | '049' | '050' | '051' | '052' | '053' | '054' | '055' | '056' | '057' |
'058' | '059' | '060' | '061' | '062' | '063' | '064' | '065' | '066' | '067' |
'068' | '069' | '070' | '071' | '072' | '073' | '074' | '075' | '076' | '077' |
'078' | '079' | '080' | '081' | '082' | '083' | '084' | '085' | '086' | '087' |
'088' | '089' | '090' | '091' | '092' | '093' | '094' | '095' | '096' | '097' |
'098' | '099' | '100' | '101' | '102' | '103' | '104' | '105' | '106' | '107' |
'108' | '109' | '110' | '111' | '112' | '113' | '114' | '115' | '116' | '117' |
'118' | '119' | '120' | '121' | '122' | '123' | '124' | '125' | '126' | '127' |
'128' | '129' | '130' | '131' | '132' | '133' | '134' | '135' | '136' | '137' |
'138' | '139' | '140' | '141' | '142' | '143' | '144' | '145' | '146' | '147' |
'148' | '149' | '150' | '151' | '152' | '153' | '154' | '155' | '156' | '157' |
'158' | '159' | '160' | '161' | '162' | '163' | '164' | '165' | '166' | '167' |
'168' | '169' | '170' | '171' | '172' | '173' | '174' | '175' | '176' | '177' |
'178' | '179' | '180' | '181' | '182' | '183' | '184' | '185' | '186' | '187' |
```

(continues on next page)

(continued from previous page)

```
'188' | '189' | '190' | '191' | '192' | '193' | '194' | '195' | '196' | '197' |
'198' | '199' | '200' | '201' | '202' | '203' | '204' | '205' | '206' | '207' |
'208' | '209' | '210' | '211' | '212' | '213' | '214' | '215' | '216' | '217' |
'218' | '219' | '220' | '221' | '222' | '223' | '224' | '225' | '226' | '227' |
'228' | '229' | '230' | '231' | '232' | '233' | '234' | '235' | '236' | '237' |
'238' | '239' | '240' | '241' | '242' | '243' | '244' | '245' | '246' | '247' |
'248' | '249' | '250' | '251' | '252' | '253' | '254' | '255' | '256' | '257' |
'258' | '259' | '260' | '261' | '262' | '263' | '264' | '265' | '266' | '267' |
'268' | '269' | '270' | '271' | '272' | '273' | '274' | '275' | '276' | '277' |
'278' | '279' | '280' | '281' | '282' | '283' | '284' | '285' | '286' | '287' |
'288' | '289' | '290' | '291' | '292' | '293' | '294' | '295' | '296' | '297' |
'298' | '299' | '300' | '301' | '302' | '303' | '304' | '305' | '306' | '307' |
'308' | '309' | '310' | '311' | '312' | '313' | '314' | '315' | '316' | '317' |
'318' | '319' | '320' | '321' | '322' | '323' | '324' | '325' | '326' | '327' |
'328' | '329' | '330' | '331' | '332' | '333' | '334' | '335' | '336' | '337' |
'338' | '339' | '340' | '341' | '342' | '343' | '344' | '345' | '346' | '347' |
'348' | '349' | '350' | '351' | '352' | '353' | '354' | '355' | '356' | '357' |
'358' | '359' | '360' | '361' | '362' | '363' | '364' | '365' | '366'

<iso8601time> ::= 'T'? <iso8601hour> (':'? <iso8601minute> (':'?
<iso8601second> (('.' | ',') <iso8601fraction>)? )? )? <iso8601timezone>?

# 24:00:00 is allowed to represent midnight at the end of a day
<iso8601hour> ::= '00' | '01' | '02' | '03' | '04' | '05' | '06' | '07' | '08'
| '09' | '10' | '11' | '12' | '13' | '14' | '15' | '16' | '17' | '18' | '19' |
'20' | '21' | '22' | '23' | '24'
<iso8601minute> ::= '00' | '01' | '02' | '03' | '04' | '05' | '06' | '07' |
'08' | '09' | '10' | '11' | '12' | '13' | '14' | '15' | '16' | '17' | '18' |
'19' | '20' | '21' | '22' | '23' | '24' | '25' | '26' | '27' | '28' | '29' |
'30' | '31' | '32' | '33' | '34' | '35' | '36' | '37' | '38' | '39' | '40' |
'41' | '42' | '43' | '44' | '45' | '46' | '47' | '48' | '49' | '50' | '51' |
'52' | '53' | '54' | '55' | '56' | '57' | '58' | '59'

# xx:yy:60 is allowed to represent leap seconds
<iso8601second> ::= '00' | '01' | '02' | '03' | '04' | '05' | '06' | '07' |
'08' | '09' | '10' | '11' | '12' | '13' | '14' | '15' | '16' | '17' | '18' |
'19' | '20' | '21' | '22' | '23' | '24' | '25' | '26' | '27' | '28' | '29' |
'30' | '31' | '32' | '33' | '34' | '35' | '36' | '37' | '38' | '39' | '40' |
'41' | '42' | '43' | '44' | '45' | '46' | '47' | '48' | '49' | '50' | '51' |
'52' | '53' | '54' | '55' | '56' | '57' | '58' | '59' | '60'
<iso8601fraction> ::= <digit>+

<iso8601timezone> ::= 'Z' | '+' <iso8601hour> (':'? <iso8601minute>)? | '-'
<iso8601hour> (':'? <iso8601minute>)?
where is_valid_iso8601datetime(str(<iso8601datetime>))

def is_valid_iso8601datetime(iso8601datetime: str) -> bool:
    """Return True iff `iso8601datetime` is valid."""
    try:
        dateutil.parser.isoparse(iso8601datetime)
        return True
    except ValueError:
        return False
```

With this, we can now create a bunch of random date/time elements:

```
$ fandango fuzz -f iso8601.fan -n 10
```

```
7093-07-04T21+13
4268-W14-1
2753
6874-07
1674
5864-06-19
2415-221T11+08
7643-W41-1
07670524T04:00+13:26
6172W21
```

And we can use additional constraints to further narrow down date intervals:

```
$ fandango fuzz -f ISO8601.fan -n 10 -c 'int(<iso8601year>) > 1950 and int(
↪<iso8601year>) < 2000'
```

```
1995-11
1984-046T18Z
1984-046
1964-046T18Z
1981-046T18Z
```

```
1964W07
1954W07
1976203
1984W07
1981W07
```

Or produce individual elements, again with individual constraints:

```
$ fandango fuzz -f ISO8601.fan -n 10 --start-symbol='<iso8601time>' -c '<iso8601hour>␣
↪== "00"'
```

```
fandango:WARNING: Symbol <start> defined, but not used
```

```
T00:18
T00
00
005538,148
0036:28.8
T00Z
00:5412
00:28-00
000434
T001856
```

Try out more constraints for yourself! The generated `ISO8601.fan` file is available for download.

CHAPTER

EIGHTEEN

PARSING AND CHECKING INPUTS

Fandango can also use its specifications to *parse* given inputs and to *check* if they conform to the specification - both

- *syntactically* (according to the grammar); and
- *semantically* (according to the constraints).

18.1 The `parse` command

To parse an existing input, Fandango provides a `parse` command. Its arguments are any *files* to be parsed; if no files are given, `parse` reads from standard input. As with the `fuzz` command, providing a specification (with `-f FILE.fan`) is mandatory.

Let us use `parse` to check some dates against the *ISO 8601 format* (page 85) we have written a Fandango spec for. The command `echo -n` outputs the string given as argument (`-n` suppresses the newline it would normally produce); the pipe symbol `|` feeds this as input into Fandango:

```
$ echo -n '2025-01-27' | fandango parse -f iso8601.fan
```

If we do this, nothing happens. That is actually a good sign: it means that Fandango has successfully parsed the input.

If we pass an *invalid* input, however, Fandango will report this. This holds for *syntactically* invalid inputs:

```
$ echo -n '01/27/2025' | fandango parse -f iso8601.fan
```

```
FandangoParseError: '<stdin>', line 1, column 4: mismatched input '2'
FandangoParseError: 1 error(s) during parsing
```

And also for *semantically* invalid inputs:

```
$ echo -n '2025-02-29' | fandango parse -f iso8601.fan
```

```
FandangoError: '<stdin>': constraint is_valid_iso8601datetime(str(<iso8601datetime>
 ↪)) not satisfied
FandangoParseError: 1 error(s) during parsing
```

In both cases, the return code will be non-zero:

```
$ echo $?
1
```

18.2 Validating Parse Results

By default, the `parse` command produces no output. However, to inspect the parse results, you can output the parsed string again. The `-o FILE` option writes the parsed string to `FILE`, with `-` being the standard output.

```
$ echo -n '2025-01-27' | fandango parse -f iso8601.fan -o -
```

```
2025-10-27
```

We see that input and output are identical (as should always be with parsing and unparsing).

Tip

As it comes to producing and storing outputs, the `parse` command has the same options as the `fuzz` command.

Since parsing and unparsing should always be symmetrical to each other, Fandango provides a `--validate` option to run this check automatically:

```
$ echo -n '2025-01-27' | fandango parse -f iso8601.fan --validate
```

Again, if nothing happens, then the (internal) check was successful.

The `--validate` option can also be passed to the `fuzz` command; here, it ensures that the produced string can be parsed by the same grammar (again, as should be).

Tip

If you find that `--validate` fails, please report this as a Fandango bug.

18.3 Alternate Output Formats

In order to debug grammars, Fandango provides a number of *alternate* formats in which to output the parsed tree, controlled by the `--format` flag.

18.3.1 String

The option `--format=string` outputs the parsed tree as a string. This is the default.

```
$ echo -n '2025-01-27' | fandango parse -f iso8601.fan -o - --format=string
```

```
2025-10-27
```

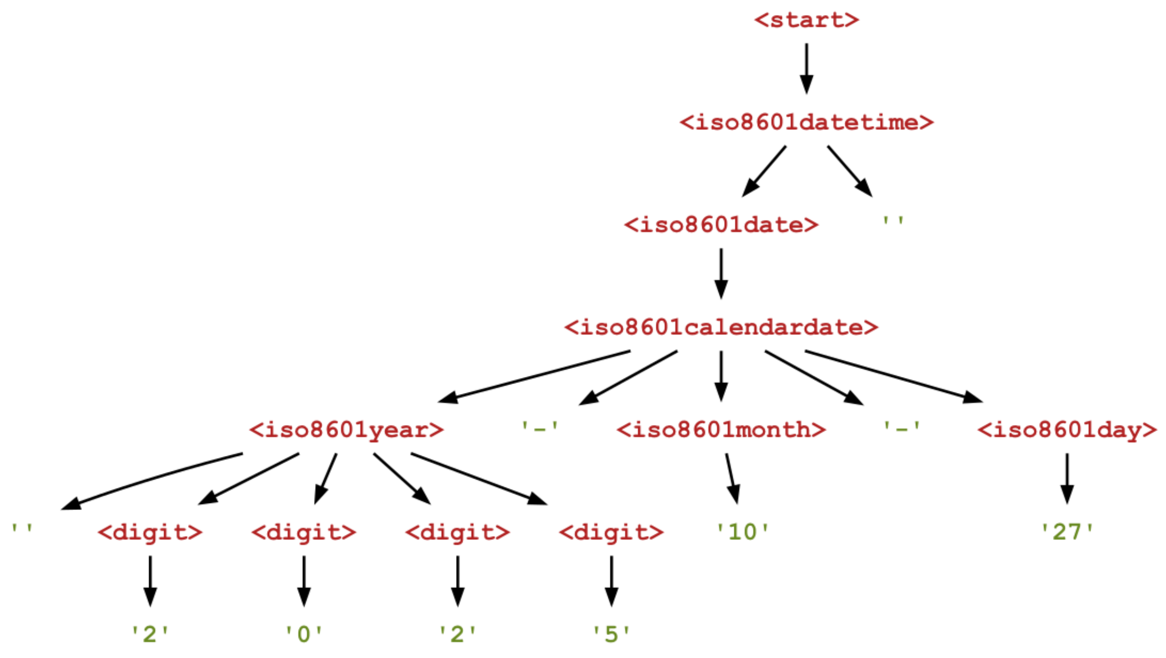
18.3.2 Tree

The option `--format=string` outputs the parsed tree as a Python `Tree()` expression. This is useful for evaluating and visualizing the tree.

```
$ echo -n '2025-01-27' | fandango parse -f iso8601.fan -o - --format=tree
```

```
Tree('<start>', Tree('<iso8601datetime>',
  Tree('<iso8601date>', Tree('<iso8601calendardate>',
    Tree('<iso8601year>',
      Tree(''),
      Tree('<digit>', Tree('<_digit>', Tree('2'))),
      Tree('<digit>', Tree('<_digit>', Tree('0'))),
      Tree('<digit>', Tree('<_digit>', Tree('2'))),
      Tree('<digit>', Tree('<_digit>', Tree('5')))
    ),
    Tree('-'),
    Tree('<iso8601month>', Tree('10')),
    Tree('-'),
    Tree('<iso8601day>', Tree('27'))
  )),
  Tree('')
))
```

Here comes this tree, visualized:



18.3.3 Grammar

The option `--format=grammar` outputs the parsed tree as a (highly specialized) grammar, in which children are indented under their respective parents. This is useful for debugging, but also for creating a grammar from a sample file and then generalizing it.

```
$ echo -n '2025-01-27' | fandango parse -f iso8601.fan -o - --format=grammar
```

```
<start> ::= <iso8601datetime>
  <iso8601datetime> ::= <iso8601date> ''  # Position 0x0000 (0); '2025-10-27'
    <iso8601date> ::= <iso8601calendardate>
      <iso8601calendardate> ::= <iso8601year> '-' <iso8601month> '-' <iso8601day>
↪# Position 0x0000 (0); '2025-10-27'
        <iso8601year> ::= '' <digit> <digit> <digit> <digit>  # Position 0x0002␣
↪(2); '2025'
          <digit> ::= <_digit>
            <_digit> ::= '2'  # Position 0x0002 (2)
          <digit> ::= <_digit>
            <_digit> ::= '0'  # Position 0x0003 (3)
          <digit> ::= <_digit>
            <_digit> ::= '2'  # Position 0x0004 (4)
          <digit> ::= <_digit>
            <_digit> ::= '5'  # Position 0x0005 (5)
        <iso8601month> ::= '10'  # Position 0x0006 (6)
        <iso8601day> ::= '27'  # Position 0x0008 (8)
```

18.3.4 Bits

The option `--format=bits` outputs the parsed tree as a bit sequence.

```
$ echo -n '2025-01-27' | fandango parse -f iso8601.fan -o - --format=bits
```

```
0011001000110000001100100011010100101101001100010011000000101101001100100011 0111
```

This is useful for debugging *binary formats* (page 101) that contain *bits* (page 117).

CHAPTER

NINETEEN

GENERATING BINARY INPUTS

Creating *binary* inputs with Fandango is a bit more challenging than creating human-readable inputs. This is because they have a few special features, such as *checksums* and *length encodings*. Fortunately, we can address all of them with dedicated constraints.

19.1 Checksums

Strictly speaking, this only holds for context-free grammars Fandango uses. *Context-sensitive* and *universal* grammars can perform arithmetic computations, but someone would have to implement them all.

Our first challenge is *checksums*. Binary input formats frequently use checksums to ensure integrity. The problem is that checksums cannot be expressed in a grammar alone, as grammars lack the arithmetic functions required to compute and check checksums. In Fandango, though, we can express the computation of a checksum in a dedicated function, which is then used in a dedicated constraint.

As an example for checksums, let's have a look at *credit card numbers*. These are definitely very human-readable and not binary at all, but for an example, they will do fine. A credit card number consists of a series of digits, where the last one is a *check digit*. Here is a grammar that expresses the structure for 16-digit credit card numbers:

```
<start>              ::= <credit_card_number>
<credit_card_number> ::= <number> <check_digit>
<number>             ::= <digit>{15}  # for 16-digit numbers
<check_digit>        ::= <digit>
```

All credit cards use Luhn's algorithm[23] to compute the check digit. Here is an implementation, adapted from the Faker library[24]. The function `credit_card_check_digit()` gets all numbers of a credit card (except the last digit) and returns the computed check digit.

```
def credit_card_check_digit(number: str) -> str:
    """Create a check digit for the credit card number `number`."""
    luhn_lookup = {
        "0": 0,
        "1": 2,
        "2": 4,
        "3": 6,
```

(continues on next page)

[23] https://en.wikipedia.org/wiki/Luhn_algorithm
[24] https://github.com/joke2k/faker/blob/master/faker/providers/credit_card/__init__.py#L99

(continued from previous page)

```
        "4": 8,
        "5": 1,
        "6": 3,
        "7": 5,
        "8": 7,
        "9": 9,
    }

    # Calculate sum
    length = len(number) + 1
    reverse = number[::-1]
    tot = 0
    pos = 0
    while pos < length - 1:
        tot += luhn_lookup[reverse[pos]]
        if pos != (length - 2):
            tot += int(reverse[pos + 1])
        pos += 2

    # Calculate check digit
    check_digit = (10 - (tot % 10)) % 10
    return str(check_digit)
```

We can easily make use of `credit_card_check_digit()` in a constraint that ties `<check_digit>` and `<number>`:

```
where <check_digit> == credit_card_check_digit(str(<number>))
```

All of this can go into a single `.fan` file: `credit_card.fan` joins the above grammar, the `credit_card_check_digit()` definition, and the above constraint into a single file.

Do not use such numbers to test third-party systems.

We can now use `credit-card.fan` to produce valid credit card numbers:

```
$ fandango fuzz -f credit_card.fan -n 10
```

```
4063468865296896
1203879705929396
3824336130241827
8266406421588048
2726941084294341
9455648412481324
1734789110984589
9906776922444485
1161878011905991
4547196313859835
```

We can also use the grammar to *parse* and *check* numbers. This credit card number should be valid:

```
$ echo -n 4931633575526870 | fandango parse -f credit_card.fan
$ echo $?  # print exit code
```

(continues on next page)

(continued from previous page)

```
0
```

Adding 1 to this number should make it *invalid*:

```
$ echo -n 4931633575526871 | fandango parse -f credit_card.fan
```

```
FandangoError: '<stdin>': constraint <check_digit> == credit_card_check_digit(str(
↪<number>)) not satisfied
FandangoParseError: 1 error(s) during parsing
```

```
$ echo $?  # print exit code
1
```

You can also simply do an Internet search for a Python implementation of the respective algorithm. Or ask your favorite AI assistant.

Similarly, you can define any kind of checksum function and then use it in a constraint. In Python, it is likely that someone has already implemented the specific checksum function, so you can also *import* it:

- The `hashlib` module[25] provides hash functions such as MD5 or SHA-256.
- The `binascii` module[26] offers CRC checks.
- The `zlib` module[27] provides CRC32 and ADLER32 checks used in zip files.

19.2 Characters and Bytes

The second set of features one frequently encounters in binary formats is, well, *bytes*. So far, we have seen Fandango operates on strings of Unicode *characters*, which use UTF-8 encoding. This clashes with a byte interpretation as soon as the produced string contains a UTF-8 prefix byte, such as `\xc2` or `\xe0`, which mark the beginning of a two- and three-byte UTF-8 sequence, respectively.

To ensure bytes will be interpreted as bytes (and as bytes only), place a `b` (binary) prefix in front of them. This ensures that a byte `b'\xc2'` will always be interpreted as a single byte, whereas will be interpreted as a single character (despite occupying multiple bytes).

Tip

Fandango provides a `<byte>` symbol by default, which expands into all bytes `b'\x00'..b'\xff'`.

[25] https://docs.python.org/3/library/hashlib.html
[26] https://docs.python.org/3/library/binascii.html
[27] https://docs.python.org/3/library/zlib.html

19.2.1 Text Files and Binary Files

By default, Fandango will read and write files in `text` mode, meaning that characters will be read in using UTF-8 encoding. However, if a grammar can produce bytes (or *bits* (page 117)), the associated files will be read and written in `binary` mode, reading and writing *bytes* instead of (encoded) characters. If your grammar contains bytes *and* strings, then the strings will be written in UTF-8 encoding into the binary file.

You can enforce a specific behavior using the Fandango `--file-mode` flag for the `fuzz` and `parse` commands:

- `fuzz --file-mode=text` opens all files in `text` mode. Strings and bytes will be written in UTF-8 encoding.
- `fuzz --file-mode=binary` opens all files in `binary` mode. Strings will be written in UTF-8 encoding; bytes will be written as is.

The default is `fuzz --file=mode=auto` (default), which will use `binary` or `text` mode as described above.

Tip

Avoid mixing non-ASCII strings with bits and bytes in a single grammar.

19.2.2 Bytes and Regular Expressions

Fandango also supports *regular expressions* (page 61) over bytes. To obtain a regular expression over a byte string, use both `r` and `b` prefixes. This is especially useful for character classes.

Here is an example: `binfinity.fan` produces strings of five bytes *outside* the range `\x80-\xff`:

```
<start> ::= rb"[^\x80-\xff]{5}"
```

This is what we get:

```
$ fandango fuzz -f binfinity.fan -n 10
```

```
0k9QM
FUkoM
\N(XZ
hA6au
_Y.[C
WNj9B
z^iG4
i:0GM
^@Qso
JvUs.
```

19.3 Length Encodings

The third set of features one frequently encounters in binary formats is *length encodings* - that is, a particular field holds a value that represents the length of one or more fields that follow. Here is a simple grammar that expresses this characteristic: A `<field>` has a two-byte length, followed by the actual (byte) content (of length `<length>`).

```
<start>    ::= <field>
<field>    ::= <length> <content>
```

(continues on next page)

(continued from previous page)

```
<length>   ::= <byte> <byte>
<content>  ::= <byte>+
```

19.3.1 Encoding Lengths with Constraints

The relationship between `<length>` and `<content>` can again be expressed using a constraint. Let us assume that `<length>` comes as a two-byte (16-bit) unsigned integer with *little-endian* encoding - that is, the low byte comes first, and the high byte follows. The value 258 (hexadecimal `0x0102`) would thus be represented as the two bytes `\x02` and `\x01`.

We can define a function `uint16()` that takes an integer and converts it to a two-byte string according to these rules. The Python method `N.to_bytes(LENGTH, ENDIANNESS)` converts the integer `N` into a bytes string of length `LENGTH`. `ENDIANNESS` is either `'big'` (default) or `'little'`.

```
def uint16(n: int) -> bytes:
    return n.to_bytes(2, 'little')
```

Using `uint16()`, we can now define how the value of `<length>` is related to the length of `<content>`:

```
where <length> == uint16(len(bytes(<content>)))
```

Tip

Having a derived value (like `<length>`) isolated on the left-hand side of an equality equation makes it easy for Fandango to first compute the content and then compute and assign the derived value.

Again, all of this goes into a single `.fan` file: `binary.fan` holds the grammar, the `uint16()` definition, and the constraint. Let us produce a single output using `binary.fan` and view its (binary) contents, using `od -c`:

```
$ fandango fuzz -n 1 -f binary.fan -o - | hexdump -C
```

```
fandango:WARNING: Could not generate a full population of unique individuals.␣
↪Population size reduced to 5.
```

```
00000000  02 00 02 00                                       |....|
00000004
```

The hexadecimal dump shows that the first two bytes encode the length of the string of digits that follows. The format is correct - we have successfully produced a length encoding.

19.3.2 Encoding Lengths with Repetitions

Another way to implement length constraints is by using *repetitions*. In Fandango, repetitions `{}` can also contain *expressions*, and like constraints, these can also refer to nonterminals that have already been parsed or produced. Hence, we can specify a rule

```
<content> ::= <byte>{f(<length>)}
```

where `f()` is a function that computes the number of `<byte>` repetitions based on `<length>`.

Let us define a variant `binary-rep.fan` that makes use of this. Here, we specify that `<content>` consists of `N` bytes, where `N` is given as follows:

We use a generator expression `:= VALUE` to prevent generated values from getting too large.

```
<start>     ::= <field>
<field>     ::= <length> <content>
<length>    ::= <byte> <byte> := b'\x00\x04'
<content>   ::= <byte>{from_uint16(<length>.value())}
```

The method `<length>.value()` returns the bytes string value of the `<length>` element. The function `from_uint16()` is defined as follows:

```
def from_uint16(n: bytes) -> int:
    return n[0] << 8 | n[1]
```

With this, we can easily produce length-encoded inputs:

```
$ fandango fuzz -n 1 -f binary-rep.fan -o - | hexdump -C
```

```
00000000  00 04 42 02 c0 2a                                 |..B..*|
00000006
```

Tip

When *parsing* (page 97) inputs, computed repetitions are much more efficient than constraints.

19.4 Converting Values to Binary Formats

Instead of implementing `uint16()` manually, we can also use the Python `struct` module[28], which offers several functions to convert data into binary formats. Using `struct`, we can redefine `uint16()` as

```
from struct import pack

def uint16(n: int) -> str:
    return pack('<H', n).decode('iso8859-1')  # convert to string
```

and obtain the same result:

```
fandango:WARNING: <field>: Concatenating 'bytes' (<length>) and 'str' (<content>)
fandango:WARNING: <length> == uint16(len(<content>)): '==': Cannot compare 'bytes'␣
↪and 'str'
```

```
ValueError: I/O operation on closed file.
0000000  005  \0   7   1   7   9   2
0000007
```

[28] https://docs.python.org/3/library/struct.html

Note that the return value of `struct.pack()` has the type `bytes` (byte string), which is different from the `str` Unicode strings that Fandango uses:

```python
pack('<H', 20)
```

```
b'\x14\x00'
```

```python
type(pack('<H', 20))
```

```
bytes
```

In Python, comparisons of different types always return `False`:

```python
# Left hand is byte string, right hand is Unicode string
b'\x14\x00' == '\x14\x00'
```

```
False
```

Hence, a constraint that compares a Fandango symbol against a byte string *will always fail*.

Warning

When comparing symbols against values, always be sure to convert the values to the appropriate type first.

Tip

Using the `'iso8859-1'` encoding (also known as `'latin-1'`) allows a 1:1 conversion of byte strings with values `'\x00'..'\xff'` into Unicode `str` strings without further interpretation.

Tip

Adding *type annotations* to functions in `.fan` files allows for future static type checking and further optimizations.

Check out the `struct` module[29] for additional encodings, including float types, long and short integers, and many more.

[29] https://docs.python.org/3/library/struct.html

CHAPTER

TWENTY

DATA CONVERTERS

When defining a complex input format, some parts may be the result of applying an *operation* on another, more structured part. Most importantly, content may be *encoded*, *compressed*, or *converted*.

Fandango uses a special form of *generators* (page 53) to handle these, called *converters*. These are generator expressions with *symbols*, mostly functions that take symbols as arguments. Let's have a look at how these work.

20.1 Encoding Data During Fuzzing

In Fandango, a *generator* (page 53) expression can contain *symbols* (enclosed in `<...>`) as elements. Such generators are called *converters*. When fuzzing, converters have the effect of Fandango using the grammar to

- instantiate each symbol from the grammar,
- evaluate the resulting expression, and
- return the resulting value.

Here is a simple example to get you started. The Python `base64` module[30] provides methods to encode arbitrary binary data into printable ASCII characters:

```
import base64

encoded = base64.b64encode(b'Fandango\x01')
encoded
```

```
b'RmFuZGFuZ28B'
```

Of course, these can be decoded again:

```
base64.b64decode(encoded)
```

```
b'Fandango\x01'
```

Let us make use of these functions. Assume we have a `<data>` field that contains a number of bytes:

```
<data> ::= b'Fandango' <byte>+
```

To encode such a `<data>` field into an `<item>`, we can write

```
<item> ::= rb'.*' := base64.b64encode(bytes(<data>))
```

[30] https://docs.python.org/3/library/base64.html

This rule brings multiple things together:

- First, we convert `<data>` into a suitable type (in our case, `bytes`).
- Then, we invoke `base64.b64encode()` on it as a generator to obtain a string of bytes.
- We parse the string into an `<item>`, whose definition is `rb'.*'` (any sequence of bytes except newline).

In a third step, we embed the `<item>` into a (binary) string:

```
<start> ::= b'Data: ' <item>
```

The full resulting `encode.fan` spec looks like this:

```
import base64

<start> ::= b'Data: ' <item>
<item> ::= rb'.*' := base64.b64encode(bytes(<data>))
<data> ::= b'Fandango' <byte>+
```

With this, we can encode and embed binary data:

```
$ fandango fuzz -f encode.fan -n 1
```

```
Data: RmFuZGFuZ29Nyhg=
```

In the same vein, one can use functions for compressing data or any other kind of conversion.

20.2 Sources, Encoders, and Constraints

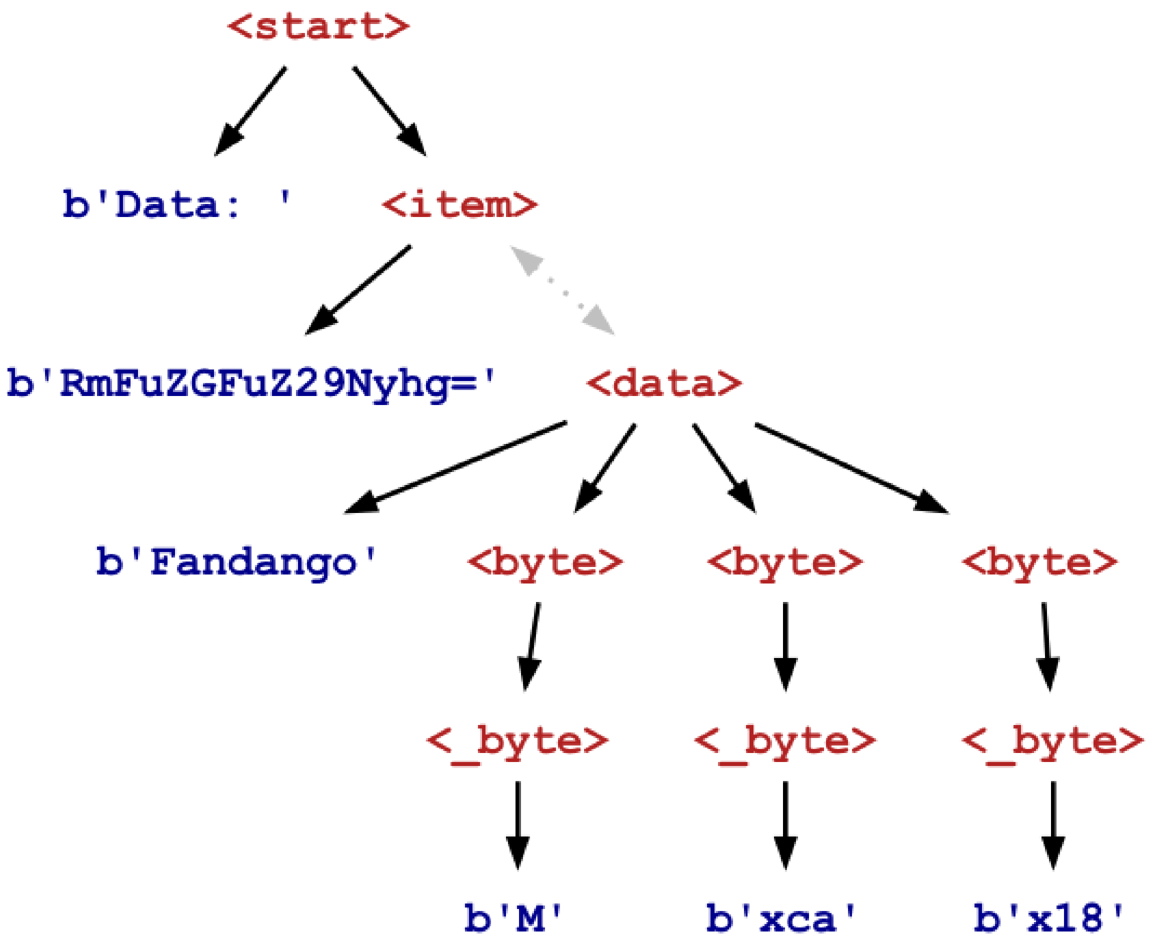
When Fandango produces an input using a generator, it *saves* the generated arguments as a *source* in the produced derivation tree. Sources become visible as soon as the input is shown as a grammar:

```
$ fandango fuzz -f encode.fan -n 1 --format=grammar
```

```
<start> ::= b'Data: ' <item>  # Position 0x0000 (0); b'Data: RmFuZGFuZ29Nyhg='
  <item> ::= b'RmFuZGFuZ29Nyhg=' := f(<data>)  # Position 0x0006 (6)
    <data> ::= b'Fandango' <byte> <byte> <byte>  # Position 0x0000 (0); b
↪'FandangoM\xca\x18'
      <byte> ::= <_byte>
        <_byte> ::= b'M'  # Position 0x0008 (8)
      <byte> ::= <_byte>
        <_byte> ::= b'\xca'  # Position 0x0009 (9)
      <byte> ::= <_byte>
        <_byte> ::= b'\x18'  # Position 0x000a (10)
```

In the definition of `<item>`, we see a generic converter `f(<data>)` as well as the definition of `<data>` that went into the generator. (The actual generator code, `base64.b64encode(bytes(<data>))`, is not saved in the derivation tree.)

We can visualize the resulting tree, using a double arrow between `<item>` and its source `<data>`, indicating that their values depend on each other:



Since sources like `<data>` are preserved, we can use them in *constraints* (page 35). For instance, we can produce a string with specific values for `<data>`:

```
$ fandango fuzz -f encode.fan -n 1 -c '<data> == b"Fandango author"'
```

```
Data: RmFuZGFuZ28gYXV0aG9y
```

Is this string a correct encoding of a correct string? We will see in the next section.

20.3 Decoding Parsed Data

So far, we can only *encode* data during fuzzing. But what if we also want to *decode* data, say during *parsing* (page 97)? Our `encode.fan` will help us *parse* the data, but not decode it:

```
$ echo -n 'Data: RmFuZGFuZ28gYXV0aG9y' | fandango parse -f encode.fan
```

```
FandangoValueError: <data>: Missing converter from <item> (<data> ::= ... := f(
↪<item>))
FandangoParseError: 1 error(s) during parsing
```

The fact that parsing fails is not a big surprise, as we only have specified an *encoder*, but not a *decoder*. As the error message suggests, we need to add a generator for `<data>` - a decoder that converts `<item>` elements into `<data>`.

We can achieve this by providing a generator for `<data>` that builds on `<item>`:

```
<data> ::= b'Fandango' <byte>+ := base64.b64decode(bytes(<item>))
```

Here, `base64.b64decode(bytes(<item>))` takes an `<item>` (which is previously parsed) and decodes it. The decoded result is parsed and placed in `<data>`.

The resulting `encode-decode.fan` file now looks like this:

```
import base64

<start> ::= b'Data: ' <item>
<item> ::= rb'.*' := base64.b64encode(bytes(<data>))
<data> ::= b'Fandango' <byte>+ := base64.b64decode(bytes(<item>))
```

Fandango allows generators in both directions so one `.fan` file can be used for fuzzing and parsing.

If this looks like a mutual recursive definition, that is because it is. During fuzzing and parsing, Fandango tracks the *dependencies* between generators and uses them to decide which generators to use first:

- When fuzzing, Fandango operates *top-down*, starting with the topmost generator encountered; their arguments are *produced*. In our case, this is the `<item>` generator, generating a value for `<data>`.
- When parsing, Fandango operates *bottom-up*, starting with the lowest generators encountered; their arguments are *parsed*. In our case, this is the `<data>` generator, parsing a value for `<item>`.

In both case, when Fandango encounters a recursion, *it stops evaluating the generator*:

- When parsing an `<item>`, Fandango does not invoke the generator for `<data>` because `<data>` is being processed already.
- Likewise, when producing `<data>`, Fandango does not invoke the generator for `<item>` because `<item>` is being processed already.

Let us see if all of this works and if this input is indeed properly parsed and decoded.

```
$ echo -n 'Data: RmFuZGFuZ28gYXV0aG9y' | fandango parse -f encode-decode.fan -o - --
↪format=grammar
```

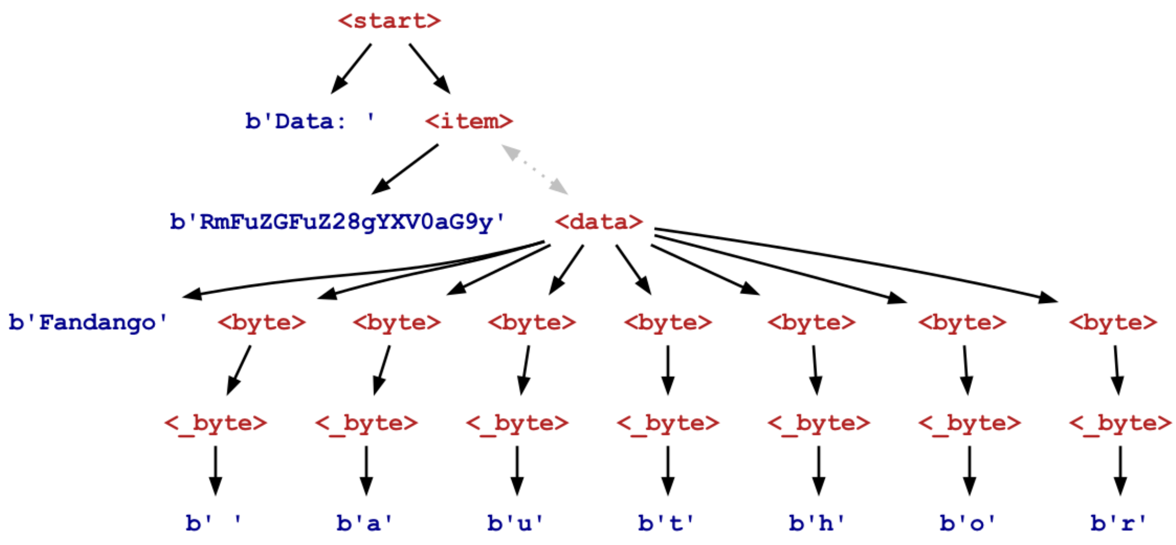
```
<start> ::= b'Data: ' <item>  # Position 0x0000 (0); b'Data: RmFuZGFuZ28gYXV0aG9y'
  <item> ::= b'RmFuZGFuZ28gYXV0aG9y' := f(<data>)  # Position 0x0006 (6)
    <data> ::= b'Fandango' <byte> <byte> <byte> <byte> <byte> <byte> <byte>  #␣
↪Position 0x0000 (0); b'Fandango author'
      <byte> ::= <_byte>
        <_byte> ::= b' '  # Position 0x0008 (8)
      <byte> ::= <_byte>
        <_byte> ::= b'a'  # Position 0x0009 (9)
      <byte> ::= <_byte>
        <_byte> ::= b'u'  # Position 0x000a (10)
      <byte> ::= <_byte>
        <_byte> ::= b't'  # Position 0x000b (11)
      <byte> ::= <_byte>
```

(continues on next page)

(continued from previous page)

```
      <_byte> ::= b'h'  # Position 0x000c (12)
    <byte> ::= <_byte>
      <_byte> ::= b'o'  # Position 0x000d (13)
    <byte> ::= <_byte>
      <_byte> ::= b'r'  # Position 0x000e (14)
```

We see that the `<data>` element contains the `"Fandango author"` string we provided as a constraint during generation. This is what the parsed derivation tree looks like:



With a constraint, we can check that the decoded string is correct:

```
$ echo -n 'Data: RmFuZGFuZ28gYXV0aG9y' | fandango parse -f encode-decode.fan -c '
↪<data> == b"Fandango author"'
```

We get no error - so the parse was successful, and that all constraints hold.

20.4 Applications

The above scheme can be used for all kinds of encodings and compressions - and thus allow *translations between abstraction layers*. Typical applications include:

- *Compressed* data (e.g. pixels in a GIF or PNG file)
- *Encoded* data (e.g. binary input as ASCII chars in MIME encodings)
- *Converted* data (e.g. ASCII to UTF-8 to UTF-16 and back)

Even though parts of the input are encoded (or compressed), you can still use *constraints* to shape them. And if the encoding or compression can be inverted, you can also use it to *parse* inputs again.

20.5 Converters vs. Constraints

Since converters (and generally, generators) can do anything, they can be used for any purpose, including producing solutions that normally would come from *constraints* (page 35).

As an example, consider the credit card grammar from the *chapter on binary inputs* (page 101):

```
<start>              ::= <credit_card_number>
<credit_card_number> ::= <number> <check_digit>
<number>             ::= <digit>{15}  # for 16-digit numbers
<check_digit>        ::= <digit>
where <check_digit> == credit_card_check_digit(str(<number>))
```

Instead of having a constraint (`where`) that expresses the relationship between `<number>` and `<check_digit>`, we can easily enhance the grammar with converters between `<number>` and `<credit_card_number>`:

```
<credit_card_number> ::= <number> <check_digit> := add_check_digit(str(<number>))
<number>             ::= <digit>{15} := strip_check_digit(str(<credit_card_number>))
```

with

```
def add_check_digit(number: str) -> str:
    """Add a check digit to the credit card number `number`."""
    check_digit = credit_card_check_digit(number)
    return number + check_digit
```

and

```
def strip_check_digit(number: str) -> str:
    """Strip the check digit from the credit card number `number`."""
    return number[:-1]
```

The resulting `.fan` spec `credit_card-gen.fan` has the same effect as the original `credit_card.fan` from the *chapter on binary inputs* (page 101):

```
$ fandango fuzz -f credit_card-gen.fan -n 10
```

```
6148894544872365
6339724979323528
0778584107671086
8483319355187389
9308372779206930
0375166395792407
5917546839683367
2773519021395108
4346627423820583
8705339260254464
```

Now, these two functions `add_check_digit()` and `strip_check_digit()` are definitely longer than our original constraint

```
where <check_digit> == credit_card_check_digit(str(<number>))
```

However, they are not necessarily more complex. And they are more efficient, as they provide a solution right away. So when should one use constraints, and when converters?

Tip

In general:

- If you have a simple, *operational* way to solve a problem, consider a *converter*.
- If you want a simple, *declarative* way to specify your needs, use a *constraint*.

CHAPTER

TWENTYONE

BITS AND BIT FIELDS

Some binary inputs use individual *bits* to specify contents. For instance, you might have a `flag` byte that holds multiple (bit) flags:

```
struct {
  unsigned int italic: 1;   // 1 bit
  unsigned int bold: 1;
  unsigned int underlined: 1;
  unsigned int strikethrough: 1;
  unsigned int brightness: 4;   // 4 bits
} format_flags;
```

How does one represent such *bit fields* in a Fandango spec?

21.1 Representing Bits

In Fandango, bits can be represented in Fandango using the special values `0` (for a zero bit) and `1` (for a non-zero bit). Hence, you can define a `<bit>` value as

```
<bit> ::= 0 | 1
```

With this, the above `format_flag` byte would be specified as

```
<start>         ::= <format_flag>
<format_flag>   ::= <italic> <bold> <underlined> <strikethrough> <brightness>
<italic>        ::= <bit>
<bold>          ::= <bit>
<underlined>    ::= <bit>
<strikethrough> ::= <bit>
<brightness>    ::= <bit>{4}
<bit>           ::= 0 | 1
```

A `<format_flag>` symbol would thus always consist of these eight bits. We can use the special option `--format=bits` to view the output as a bit stream:

```
$ fandango fuzz --format=bits -f bits.fan -n 1 --start-symbol='<format_flag>'
```

```
fandango:WARNING: Symbol <start> defined, but not used
10110010
```

Note

The combination of `--format=bits` and `--start-symbol` is particularly useful to debug bit fields.

Internally, Fandango treats individual flags as integers, too. Hence, we can also apply *constraints* to the individual flags. For instance, we can profit from the fact that Python treats `0` as False and `1` as True:

```
$ fandango fuzz --format=bits -f bits.fan -n 10 -c '<italic> and <bold>'
```

```
11011111
11011010
11011001
11100110
11000001
11001000
11111101
11110001
11100010
11100011
```

Fandango strictly follows a "left-to-right" order - that is, the order in which bits and bytes are specified in the grammar, the most significant bit is stored first.

Hence, we can also easily set the value of the entire `brightness` field using a constraint:

```
$ fandango fuzz --format=bits -f bits.fan -n 1 -c '<brightness> == 0b1111'
```

```
11011111
```

Note

Fandango always strictly follows a "left-to-right" order - that is, the order in which bits and bytes are specified in the grammar.

Of course, we can also give the number in decimal format:

```
$ fandango fuzz --format=bits -f bits.fan -n 1 -c '<brightness> == 15'
```

```
11111111
00011111
11001111
00001111
01001111
01101111
10111111
10101111
11011111
11101111
```

Note how the last four bits (the `<brightness>` field) are always set to `1111` - the number 15.

Warning

When implementing a format, be sure to follow its conventions regarding

- *bit ordering* (most or least significant bit first)
- *byte ordering* (most or least significant byte first)

21.2 Parsing Bits

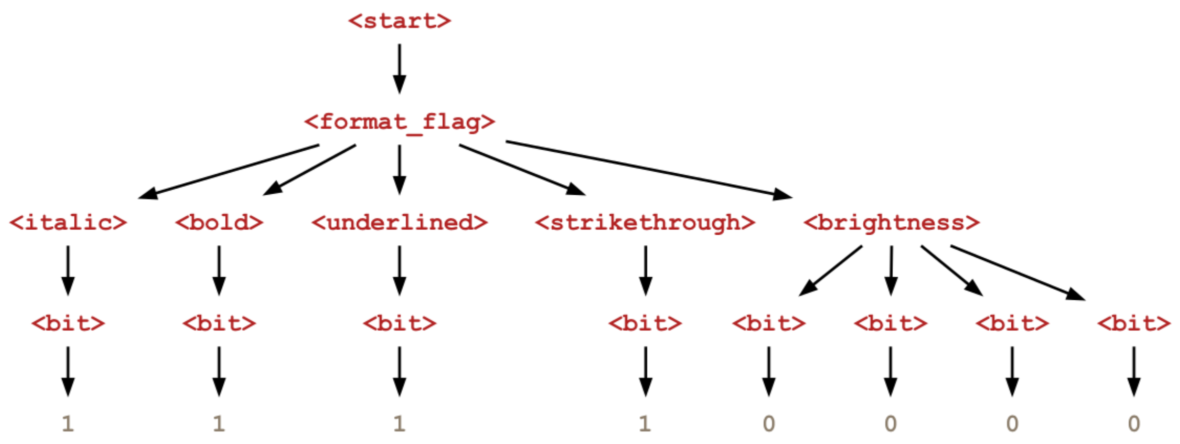
Fandango also supports *parsing* (page 97) inputs with bits. This is what happens if we send a byte `\xf0` (the upper four bits set) to the parser:

```
$ echo -n '\xf0' | fandango parse -f bits.fan -o - --format=bits
```

```
11110000
```

We see that the input was properly parsed and decomposed into individual bits.

This is the resulting parse tree:



The `grammar` format shows us that the values are properly assigned:

```
$ echo -n '\xf0' | fandango parse -f bits.fan -o - --format=grammar
```

```
<start> ::= <format_flag>
  <format_flag> ::= <italic> <bold> <underlined> <strikethrough> <brightness>  #␣
↪0b11110000 (240)
    <italic> ::= <bit>
      <bit> ::= 1  # Position 0x0000 (0), bit 7
    <bold> ::= <bit>
      <bit> ::= 1  # Position 0x0000 (0), bit 6
    <underlined> ::= <bit>
      <bit> ::= 1  # Position 0x0000 (0), bit 5
    <strikethrough> ::= <bit>
      <bit> ::= 1  # Position 0x0000 (0), bit 4
    <brightness> ::= <bit> <bit> <bit> <bit>  # 0b0 (0)
      <bit> ::= 0  # Position 0x0000 (0), bit 3
      <bit> ::= 0  # Position 0x0000 (0), bit 2
      <bit> ::= 0  # Position 0x0000 (0), bit 1
      <bit> ::= 0  # Position 0x0000 (0), bit 0
```

Warning

To parse bits properly, they must come in multiples of eight.

21.3 Bits and Padding

When generating binary inputs, you may need to adhere to specific *lengths*. Such lengths are often enforced by *padding* – that is, adding bits until the required length is achieved. For instance, let us assume you have a field consisting of some bits. However, the overall length of the field must be a multiple of eight to have it byte-aligned. For such *padding*, define the field as

```
<field> ::= <bits> <padding>
<padding> ::= 0*
```

combined with a constraint

```
where len(<field>) % 8 == 0
```

Note that applied on derivation trees, `len()` always returns the number of child elements, not the string length; here, we use this to access the number of elements in `<field>`.

CHAPTER

TWENTYTWO

CASE STUDY: THE GIF FORMAT

Error

To be added later.

The GIF format[31] is widely used to encode image sequences.

We start with a very short GIF to keep things simple (source[32]): tinytrans.gif.

We can parse this file using Fandango:

```
!fandango parse -f gif89a.fan tinytrans.gif -o - --format=grammar --validate
assert _exit_code == 0
```

```
<start> ::= <GifHeader> <LogicalScreenDescriptor> <GlobalColorTable> <Data_1>
 ↪<Trailer>  # b'GIF89a\x01\x00\x01\x00\x80\x01\x00\xff\xff\xff\x00\x00\x00!\xf9\
 ↪x04\x01\n\x00\x01\x00,\x00\x00\x00\x00\x01\x00\x01\x00\x00\x02\x02L\x01\x00;'
  <GifHeader> ::= <GIFHEADER>
    <GIFHEADER> ::= <Signature> <Version>  # b'GIF89a'
      <Signature> ::= <char> <char> <char>  # b'GIF'
        <char> ::= <byte>
          <byte> ::= <_byte>
            <_byte> ::= b'G'  # Position 0x0000 (0)
        <char> ::= <byte>
          <byte> ::= <_byte>
            <_byte> ::= b'I'  # Position 0x0001 (1)
        <char> ::= <byte>
          <byte> ::= <_byte>
            <_byte> ::= b'F'  # Position 0x0002 (2)
      <Version> ::= <char> <char> <char>  # b'89a'
        <char> ::= <byte>
          <byte> ::= <_byte>
            <_byte> ::= b'8'  # Position 0x0003 (3)
        <char> ::= <byte>
          <byte> ::= <_byte>
            <_byte> ::= b'9'  # Position 0x0004 (4)
        <char> ::= <byte>
          <byte> ::= <_byte>
            <_byte> ::= b'a'  # Position 0x0005 (5)
  <LogicalScreenDescriptor> ::= <LOGICALSCREENDESCRIPTOR>
    <LOGICALSCREENDESCRIPTOR> ::= <Width> <Height> <PackedFields>
```

(continues on next page)

[31] https://www.fileformat.info/format/gif/egff.htm
[32] http://probablyprogramming.com/2009/03/15/the-tiniest-gif-ever

(continued from previous page)

```
↪<BackgroundColorIndex> <PixelAspectRatio>  # b'\x01\x00\x01\x00\x80\x01\x00'
     <Width> ::= b'\x01' b'\x00'  # Position 0x0006 (6); b'\x01\x00'
     <Height> ::= b'\x01' b'\x00'  # Position 0x0008 (8); b'\x01\x00'
     <PackedFields> ::= <LOGICALSCREENDESCRIPTOR_PACKEDFIELDS>
       <LOGICALSCREENDESCRIPTOR_PACKEDFIELDS> ::= <GlobalColorTableFlag>
↪<ColorResolution> <SortFlag> <SizeOfGlobalColorTable>  # 0b10000000 (128)
         <GlobalColorTableFlag> ::= 1  # Position 0x000a (10), bit 7
         <ColorResolution> ::= 0 0 0  # Position 0x000a (10), bits 6-4; 0b0 (0)
         <SortFlag> ::= <bit>
           <bit> ::= <_bit>
             <_bit> ::= 0  # Position 0x000a (10), bit 3
         <SizeOfGlobalColorTable> ::= 0 0 0  # Position 0x000a (10), bits 2-0;␣
↪0b0 (0)
     <BackgroundColorIndex> ::= b'\x01'  # Position 0x000b (11)
     <PixelAspectRatio> ::= b'\x00'  # Position 0x000c (12)
 <GlobalColorTable> ::= <RGB> b'\x00' b'\x00' b'\x00'  # Position 0x000d (13); b'\
↪xff\xff\xff\x00\x00\x00'
   <RGB> ::= <R> <G> <B>  # b'\xff\xff\xff'
     <R> ::= <UBYTE>
       <UBYTE> ::= <ubyte>
         <ubyte> ::= <uchar>
           <uchar> ::= <unsigned_char>
             <unsigned_char> ::= <byte>
               <byte> ::= <_byte>
                 <_byte> ::= b'\xff'  # Position 0x0010 (16)
     <G> ::= <UBYTE>
       <UBYTE> ::= <ubyte>
         <ubyte> ::= <uchar>
           <uchar> ::= <unsigned_char>
             <unsigned_char> ::= <byte>
               <byte> ::= <_byte>
                 <_byte> ::= b'\xff'  # Position 0x0011 (17)
     <B> ::= <UBYTE>
       <UBYTE> ::= <ubyte>
         <ubyte> ::= <uchar>
           <uchar> ::= <unsigned_char>
             <unsigned_char> ::= <byte>
               <byte> ::= <_byte>
                 <_byte> ::= b'\xff'  # Position 0x0012 (18)
 <Data_1> ::= <DATA>
   <DATA> ::= <GraphicControlExtension> <ImageDescriptor> <LocalColorTable>
↪<ImageData>  # b'!\xf9\x04\x01\n\x00\x01\x00,\x00\x00\x00\x00\x01\x00\x01\x00\
↪x00\x02\x02L\x01\x00'
     <GraphicControlExtension> ::= <GRAPHICCONTROLEXTENSION>
       <GRAPHICCONTROLEXTENSION> ::= <ExtensionIntroducer_1> <GraphicControlLabel_
↪1> <GraphicControlSubBlock> <BlockTerminator_2>  # b'!\xf9\x04\x01\n\x00\x01\x00'
         <ExtensionIntroducer_1> ::= b'!'  # Position 0x0013 (19)
         <GraphicControlLabel_1> ::= b'\xf9'  # Position 0x0014 (20)
         <GraphicControlSubBlock> ::= <GRAPHICCONTROLSUBBLOCK>
           <GRAPHICCONTROLSUBBLOCK> ::= <BlockSize> <PackedFields_2> <DelayTime>
↪<TransparentColorIndex>  # b'\x04\x01\n\x00\x01'
             <BlockSize> ::= b'\x04'  # Position 0x0015 (21)
             <PackedFields_2> ::= <GRAPHICCONTROLEXTENSION_DATASUBBLOCK_
↪PACKEDFIELDS>
               <GRAPHICCONTROLEXTENSION_DATASUBBLOCK_PACKEDFIELDS> ::= 0 0 0 0 0␣
↪0 0 1  # Position 0x0016 (22), bits 7-0; 0b1 (1)
             <DelayTime> ::= b'\n' b'\x00'  # Position 0x0017 (23); b'\n\x00'
```

(continues on next page)

(continued from previous page)

```
              <TransparentColorIndex> ::= b'\x01'  # Position 0x0019 (25)
          <BlockTerminator_2> ::= b'\x00'  # Position 0x001a (26)
      <ImageDescriptor> ::= <IMAGEDESCRIPTOR>
        <IMAGEDESCRIPTOR> ::= <ImageSeperator_1> <ImageLeftPosition>
↪<ImageTopPosition> <ImageWidth> <ImageHeight> <PackedFields_1>  # b',\x00\x00\
↪x00\x00\x01\x00\x01\x00\x00'
          <ImageSeperator_1> ::= b','  # Position 0x001b (27)
          <ImageLeftPosition> ::= b'\x00' b'\x00'  # Position 0x001c (28); b'\x00\
↪x00'
          <ImageTopPosition> ::= b'\x00' b'\x00'  # Position 0x001e (30); b'\x00\
↪x00'
          <ImageWidth> ::= b'\x01' b'\x00'  # Position 0x0020 (32); b'\x01\x00'
          <ImageHeight> ::= b'\x01' b'\x00'  # Position 0x0022 (34); b'\x01\x00'
          <PackedFields_1> ::= 0 0 0 0 0 0 0 0  # Position 0x0024 (36), bits 7-0;␣
↪0b0 (0)
      <LocalColorTable> ::= b'\x02' b'\x02' b'L'  # Position 0x0025 (37); b'\x02\
↪x02L'
      <ImageData> ::= <IMAGEDATA>
        <IMAGEDATA> ::= <LZWMinimumCodeSize> <DataSubBlocks>  # b'\x01\x00'
          <LZWMinimumCodeSize> ::= b'\x01'  # Position 0x0028 (40)
          <DataSubBlocks> ::= b'\x00'  # Position 0x0029 (41)
 <Trailer> ::= <TRAILER>
   <TRAILER> ::= <GIFTrailer_1>
     <GIFTrailer_1> ::= b';'  # Position 0x002a (42)
```

CHAPTER

TWENTYTHREE

HATCHING SPECS

Fandango provides an `include()` function that you can use to *include* existing Fandango content. This allows you to distribute specifications over multiple files, defining *base* specs whose definitions can be further *refined* in specs that use them.

23.1 Including Specs with `include()`

Specifically, in a `.fan` file, a call to `include(FILE)`

1. Finds and loads `FILE` (typically *in the same location as the including file* (page 167))
2. Executes the *code* in `FILE`
3. Parses and adds the *grammar* in `FILE`
4. Parses and adds the *constraints* in `FILE`.

The `include()` function allows for *incremental refinement* of Fandango specifications - you can create some base `base.fan` spec, and then have more *specialized* specifications that alter grammar rules, add more constraints, or refine the code.

23.2 Incremental Refinement

Let us assume you have a *base* spec for a particular format, say, `base.fan`. Then, in a *refined* spec (say, `refined.fan`) that *includes* `base.fan`, you can

- override *grammar definitions*, by redefining rules;
- override *function and constant definitions*, by redefining them; and
- add additional *constraints*.

As an example, consider our `persons.fan` *definition of a name database* (page 25). We can create a more specialized version `persons50.fan` by including `persons.fan` and adding a *constraint* (page 35):

```
include('persons.fan')
where int(<age>) < 50
```

Likewise, we can create a specialized version `persons-faker.fan` that uses *fakers* (page 53) by overriding the `<first_name>` and `<last_name>` definitions:

```
from faker import Faker
fake = Faker()

include('persons.fan')

<first_name> ::= <name> := fake.first_name()
<last_name> ::= <name> := fake.last_name()
```

The *include* mechanism thus allows us to split responsibilities across multiple files:

- We can have one spec #1 with basic definitions of individual elements
- We can have a spec #2 that uses (includes) these basic definitions from spec #1 to define a *syntax*
- We can have a spec #3 that refines spec #2 to define a specific format for a particular program or device
- We can have a spec #4 that refines spec #3 towards a particular testing goal.

These mechanisms are akin to *inheritance* and *specialization* in object-oriented programming.

Tip

Generally, Fandango will warn about unused symbols, but not in an included `.fan` file.

23.3 Crafting a Library

If you create multiple specifications, you may wonder where best to store them. The *rules for where Fandango searches for included files* (page 167) are complex, but they boil down to two simple rules:

Tip

Store your included Fandango specs either

- in the directory where the *including* specs are, or
- in `$HOME/.local/share/fandango` (or `$HOME/Library/Fandango` on a Mac).

23.4 `include()` vs. `import`

Python provides its own import mechanism for referring to existing features. In general, you should use

- `import` whenever you want to make use of Python functions; and
- `include()` only if you want to make use of Fandango features.

Warning

Using `include` for *pure Python code*, as in `include('code.py')` is not recommended. Most importantly, the current Fandango implementation will process “included” Python code only *after* all code in the “including” spec has been run. In contrast, the effects of `import` are immediate.

Part III

Fandango Reference

CHAPTER

TWENTYFOUR

FANDANGO REFERENCE

This manual contains reference information about Fandango:

CHAPTER

TWENTYFIVE

INSTALLING FANDANGO

25.1 Installing Fandango for Normal Usage

Fandango comes as a Python package. To install Fandango, run the following command:

```
$ pip install fandango-fuzzer
```

To test if everything worked well, try

```
$ fandango --help
```

which should give you a list of options:

```
usage: fandango [-h] [--version] [--verbose | --quiet]
                {fuzz,parse,shell,help,copyright,version} ...

The access point to the Fandango framework

options:
  -h, --help            show this help message and exit
  --version             show version number
  --verbose, -v         increase verbosity. Can be given multiple times (-vv)
  --quiet, -q           decrease verbosity. Can be given multiple times (-qq)

commands:
  {fuzz,parse,shell,help,copyright,version}
                        the command to execute
    fuzz                produce outputs from .fan files and test programs
    parse               parse input file(s) according to .fan spec
    shell               run an interactive shell (default)
    help                show this help and exit
    copyright           show copyright
    version             show version

Use `fandango help` to get a list of commands.
Use `fandango help COMMAND` to learn more about COMMAND.
See https://fandango-fuzzer.github.io/ for more information.
```

25.2 Installing Fandango for Development

Caution

This will get you the very latest version of Fandango, which may be unstable. Use at your own risk.

Clone the Fandango repository:

```
$ git clone https://github.com/fandango-fuzzer/fandango/
```

Switch to the top-level `fandango/` folder:

```
$ cd fandango
```

Run

```
$ pip install -e .
```

You should then be able to invoke Fandango as described above.

CHAPTER

TWENTYSIX

FANDANGO COMMAND REFERENCE

26.1 All Commands

Here is a list of all `fandango` commands:

```
usage: fandango [-h] [--version] [--verbose | --quiet]
                {fuzz,parse,shell,help,copyright,version} ...

The access point to the Fandango framework

options:
  -h, --help            show this help message and exit
  --version             show version number
  --verbose, -v         increase verbosity. Can be given multiple times (-vv)
  --quiet, -q           decrease verbosity. Can be given multiple times (-qq)

commands:
  {fuzz,parse,shell,help,copyright,version}
                        the command to execute
    fuzz                produce outputs from .fan files and test programs
    parse               parse input file(s) according to .fan spec
    shell               run an interactive shell (default)
    help                show this help and exit
    copyright           show copyright
    version             show version

Use `fandango help` to get a list of commands.
Use `fandango help COMMAND` to learn more about COMMAND.
See https://fandango-fuzzer.github.io/ for more information.
```

26.2 Fuzzing

To *produce outputs with `fandango`* (page 25), use `fandango fuzz`:

```
usage: fandango fuzz [-h] [-f FAN_FILE] [-c CONSTRAINT] [--no-cache]
                     [--no-stdlib] [-s SEPARATOR] [-I DIR] [-d DIRECTORY]
                     [-x FILENAME_EXTENSION]
                     [--format {string,bits,tree,grammar,value,repr,none}]
                     [--file-mode {text,binary,auto}] [--validate]
                     [-S START_SYMBOL] [--warnings-are-errors]
                     [-N MAX_GENERATIONS] [--population-size POPULATION_SIZE]
```

(continues on next page)

(continued from previous page)

```
                     [--elitism-rate ELITISM_RATE]
                     [--crossover-rate CROSSOVER_RATE]
                     [--mutation-rate MUTATION_RATE]
                     [--random-seed RANDOM_SEED]
                     [--destruction-rate DESTRUCTION_RATE]
                     [--max-repetition-rate MAX_REPETITION_RATE]
                     [--max-repetitions MAX_REPETITIONS]
                     [--max-node-rate MAX_NODE_RATE] [--max-nodes MAX_NODES]
                     [-n NUM_OUTPUTS] [--best-effort] [-i INITIAL_POPULATION]
                     [-o OUTPUT] [--input-method {stdin,filename}]
                     [command] ...

options:
  -h, --help            show this help message and exit
  -f FAN_FILE, --fandango-file FAN_FILE
                        Fandango file (.fan, .py) to be processed. Can be
                        given multiple times. Use '-' for stdin
  -c CONSTRAINT, --constraint CONSTRAINT
                        define an additional constraint CONSTRAINT. Can be
                        given multiple times.
  --no-cache            do not cache parsed Fandango files.
  --no-stdlib           do not use standard library when parsing Fandango
                        files.
  -s SEPARATOR, --separator SEPARATOR
                        output SEPARATOR between individual inputs. (default:
                        newline)
  -I DIR, --include-dir DIR
                        specify a directory DIR to search for included
                        Fandango files
  -d DIRECTORY, --directory DIRECTORY
                        create individual output files in DIRECTORY
  -x FILENAME_EXTENSION, --filename-extension FILENAME_EXTENSION
                        extension of generated file names (default: '.txt')
  --format {string,bits,tree,grammar,value,repr,none}
                        produce output(s) as string (default), as a bit
                        string, as a derivation tree, as a grammar, as a
                        Python value, in internal representation, or none
  --file-mode {text,binary,auto}
                        mode in which to open and write files (default is
                        'auto': 'binary' if grammar has bits or bytes, 'text'
                        otherwise)
  --validate            run internal consistency checks for debugging
  -o OUTPUT, --output OUTPUT
                        write output to OUTPUT (default: stdout)

general settings:
  -S START_SYMBOL, --start-symbol START_SYMBOL
                        the grammar start symbol (default: `<start>`)
  --warnings-are-errors
                        treat warnings as errors

algorithm settings:
  -N MAX_GENERATIONS, --max-generations MAX_GENERATIONS
                        the maximum number of generations to run the algorithm
  --population-size POPULATION_SIZE
                        the size of the population
  --elitism-rate ELITISM_RATE
```

(continues on next page)

(continued from previous page)

```
                        the rate of individuals preserved in the next
                        generation
  --crossover-rate CROSSOVER_RATE
                        the rate of individuals that will undergo crossover
  --mutation-rate MUTATION_RATE
                        the rate of individuals that will undergo mutation
  --random-seed RANDOM_SEED
                        the random seed to use for the algorithm
  --destruction-rate DESTRUCTION_RATE
                        the rate of individuals that will be randomly
                        destroyed in every generation
  --max-repetition-rate MAX_REPETITION_RATE
                        rate at which the number of maximal repetitions should
                        be increased
  --max-repetitions MAX_REPETITIONS
                        Maximal value, the number of repetitions can be
                        increased to
  --max-node-rate MAX_NODE_RATE
                        rate at which the maximal number of nodes in a tree is
                        increased
  --max-nodes MAX_NODES
                        Maximal value, the number of nodes in a tree can be
                        increased to
  -n NUM_OUTPUTS, --num-outputs NUM_OUTPUTS, --desired-solutions NUM_OUTPUTS
                        the number of outputs to produce (default: 100)
  --best-effort         produce a 'best effort' population (may not satisfy
                        all constraints)
  -i INITIAL_POPULATION, --initial-population INITIAL_POPULATION
                        directory or ZIP archive with initial population

command invocation settings:
  --input-method {stdin,filename}
                        when invoking COMMAND, choose whether Fandango input
                        will be passed as standard input (`stdin`) or as last
                        argument on the command line (`filename`) (default)
  command               command to be invoked with a Fandango input
  args                  the arguments of the command
```

26.3 Parsing

To *parse inputs with `fandango`* (page 97), use `fandango parse`:

```
usage: fandango parse [-h] [-f FAN_FILE] [-c CONSTRAINT] [--no-cache]
                      [--no-stdlib] [-s SEPARATOR] [-I DIR] [-d DIRECTORY]
                      [-x FILENAME_EXTENSION]
                      [--format {string,bits,tree,grammar,value,repr,none}]
                      [--file-mode {text,binary,auto}] [--validate]
                      [-S START_SYMBOL] [--warnings-are-errors] [--prefix]
                      [-o OUTPUT]
                      [files ...]

positional arguments:
  files                 files to be parsed. Use '-' for stdin
```

(continues on next page)

(continued from previous page)

```
options:
  -h, --help            show this help message and exit
  -f FAN_FILE, --fandango-file FAN_FILE
                        Fandango file (.fan, .py) to be processed. Can be
                        given multiple times. Use '-' for stdin
  -c CONSTRAINT, --constraint CONSTRAINT
                        define an additional constraint CONSTRAINT. Can be
                        given multiple times.
  --no-cache            do not cache parsed Fandango files.
  --no-stdlib           do not use standard library when parsing Fandango
                        files.
  -s SEPARATOR, --separator SEPARATOR
                        output SEPARATOR between individual inputs. (default:
                        newline)
  -I DIR, --include-dir DIR
                        specify a directory DIR to search for included
                        Fandango files
  -d DIRECTORY, --directory DIRECTORY
                        create individual output files in DIRECTORY
  -x FILENAME_EXTENSION, --filename-extension FILENAME_EXTENSION
                        extension of generated file names (default: '.txt')
  --format {string,bits,tree,grammar,value,repr,none}
                        produce output(s) as string (default), as a bit
                        string, as a derivation tree, as a grammar, as a
                        Python value, in internal representation, or none
  --file-mode {text,binary,auto}
                        mode in which to open and write files (default is
                        'auto': 'binary' if grammar has bits or bytes, 'text'
                        otherwise)
  --validate            run internal consistency checks for debugging
  --prefix              parse a prefix only
  -o OUTPUT, --output OUTPUT
                        write output to OUTPUT (default: none). Use '-' for
                        stdout

general settings:
  -S START_SYMBOL, --start-symbol START_SYMBOL
                        the grammar start symbol (default: `<start>`)
  --warnings-are-errors
                        treat warnings as errors
```

26.4 Shell

To *enter commands in* `fandango` (page 47), use `fandango shell` or just `fandango`:

```
usage: fandango shell [-h]

options:
  -h, --help  show this help message and exit
```

CHAPTER

TWENTYSEVEN

FANDANGO LANGUAGE REFERENCE

This chapter specifies the exact syntax (and semantics) of Fandango specifications (`.fan` files).

27.1 General Structure

We specify the Fandango syntax in form of a Fandango specification.

A `.fan` Fandango specification file consists of

- grammar productions (page 138) (`<production>`)
- constraints (page 142) (`<constraint>`)
- Python code (page 143) (`<python_statement>`).

```
<start> ::= <fandango>
<fandango> ::= <statement>*
<statement> ::= <production> | <constraint> | <python_statement> | <newline> |
↪<comment>
```

27.2 Whitespace

Besides grammar productions, constraints, and code, a `.fan` file can contain *newlines* (`<newline>`) and *whitespace* (`<_>`).

The *generators* (`:= '\n'` and `:= ' '`) specify useful default values when fuzzing.

```
<newline> ::= ('\r'? '\n' | '\r' | '\f') := '\n'
<_> ::= r'[ \t]*' := ' '
```

27.3 Physical and Logical Lines

As in Python[33], one can join two physical lines into a logical by adding a backslash \ at the end of the first line.

27.4 Comments

Comments are as in Python; they are introduced by a # character and continue until the end of the line. By convention, there are two spaces in front of the # and one space after.

```
<comment> ::= <_>{2} '#' <_> r'[^\r\n\f]'* <newline>
```

Note

The actual implementation allows a comment at any end of a line.

27.5 Grammars

A *grammar* is a set of *productions*. Each production defines a nonterminal (page 138) followed by `::=` and a number of alternatives (page 139).

An optional generator (page 142) can define a Python function to produce a value during fuzzing.

Productions end with a newline or a `;` character.

```
<production> ::= (
    <nonterminal> <_> '::=' <_> <alternatives>
    <generator>? <comment>? (';' | <newline>))
```

27.6 Nonterminals

A *nonterminal* is a Python identifier enclosed in angle brackets `<...>`. It starts with a letter (regular expression `\w`) or an underscore (_), followed by more letters, digits (regular expression `\d`), or underscores.

```
<nonterminal> ::= '<' <name> '>'
<name> ::= r'(\w|_)(\w|\d|_)*'
```

Like Python, Fandango allows all Unicode letters and digits in identifiers.

Note

For portability, we recommend to use only ASCII letters `a...z`, `A...Z`, digits `0...9`, and underscores _ in identifiers.

[33] https://docs.python.org/3/reference/lexical_analysis.html#explicit-line-joining

27.7 Alternatives

The *alternatives* part of a production rule defines possible *expansions* (page 139) (`<concatenation>`) for a nonterminal, separated by `|`.

```
<alternatives> ::= <concatenation> (<_> '|' <_> <concatenation>)*
```

27.8 Concatenations

A *concatenation* is a sequence of individual *operators* (page 139) (typically symbols such as strings).

```
<concatenation> ::= <operator> (<_> <operator>)*
```

27.9 Symbols and Operators

An operator is a *symbol* (`<symbol>`), followed by an optional *repetition* (page 139) operator.

```
<operator> ::= <symbol> | <kleene> | <plus> | <option> | <repeat>
```

A symbol can be a nonterminal (page 138), a *string* (page 140) or *bytes* (page 141) *literal*, a number (page 141) (for bits), a generator call (page 142), or (parenthesized) *alternatives* (page **??**).

```
<symbol> ::= (
      <nonterminal>
    | <string_literal>
    | <bytes_literal>
    | <number>
    | <generator_call>
    | '(' <alternatives> ')'
    )
```

27.10 Repetitions

Any symbol can be followed by a repetition specification. The syntax `{N,M}` stands for a number of repetitions from `N` to `M`. For example, `<a>{3,5}` will match from 3 to 5 `<a>` symbols.

Both `N` and `M` can be omitted:

- Omitting `N` creates a lower bound of zero.
- Omitting `M` creates an infinite upper bound (i.e, any number of repetitions).
- The comma may not be omitted, as this would create confusion with `{N}` (see below).

Tip

In Fandango, the number of repetitions is limited. Use the `--max-repetitions M` flag to change the limit.

Fandango supports a number of abbreviations for repetitions:

- The form `{N}` stands for `{N, N}` (exactly `N` repetitions)
- The form `*` stands for `{0,}` (zero or more repetitions)
- The form `+` stands for `{1,}` (one or more repetitions)
- The form `?` stands for `{0,1}` (an optional element)

```
<kleene> ::= <symbol> '*'
<plus>  ::= <symbol> '+'
<option> ::= <symbol> '?'
<repeat> ::= (
     <symbol> '{' <python_expression> '}'
   | <symbol> '{' <python_expression>? ',' <python_expression>? '}')
```

27.11 String Literals

Fandango supports the full Python syntax for string literals[34]:

- Short strings are enclosed in single (`'...'`) or double (`"..."`) quotes
- Long strings are enclosed in triple quotes (`'''...'''` and `"""..."""`)
- One can use escape sequences[35] (`\n`) to express special characters

Fandango interprets Python *raw strings* (using an `r` prefix, as in `r'foo'`) as *regular expressions*. During parsing, these are matched against the input; during fuzzing, they are instantiated into a matching string.

```
<string_literal> ::= (
      r'[rR]'
    | r'[uU]'
    | r'[fF]'
    | r'[fF][rR]'
    | r'[rR][fF]')? ( <short_string> | <long_string>)
```

```
<short_string> ::= (
      "'" (<string_escape_seq> | r"[^\\\r\n\f']")* "'"
    | '"' (<string_escape_seq> | r'[^\\\r\n\f"]')* '"')
```

```
<string_escape_seq> ::= '\\' r'.' | '\\' <newline>
```

```
<long_string> ::= (
      "'''" <long_string_item>* "'''"
    | '"""' <long_string_item>* '"""')

<long_string_item> ::= <long_string_char> | <string_escape_seq>
<long_string_char> ::= r'[^\\]'
```

[34] https://docs.python.org/3/reference/lexical_analysis.html#string-and-bytes-literals
[35] https://docs.python.org/3/reference/lexical_analysis.html#escape-sequences

27.12 Byte Literals

Byte literals (a string prefixed with `b`) are interpreted as in Python; the *rules for strings* (page **??**) apply as well.

- If a grammar can produce bytes (or *bits* (page 141)), the associated files will be read and written in `binary` mode, reading and writing *bytes* instead of (encoded) characters.
- If the grammar contains bytes *and* strings, then strings will be written in UTF-8 encoding into the binary file.
- See the *section on binary files* (page 101) for more details.

```
<bytes_literal> ::= (
      r'[bB]'
    | r'[bB][rR]'
    | r'[rR][bB]') (<short_bytes> | <long_bytes>)
```

```
<short_bytes> ::= (
      "'" (<short_bytes_char_no_single_quote> | <bytes_escape_seq>)* "'"
    | '"' ( <short_bytes_char_no_double_quote> | <bytes_escape_seq> )* '"')
```

```
<bytes_escape_seq> ::= '\\' r'[\u0000-\u007F]'
```

```
<short_bytes_char_no_single_quote> ::= (
      r'[\u0000-\u0009]'
    | r'[\u000B-\u000C]'
    | r'[\u000E-\u0026]'
    | r'[\u0028-\u005B]'
    | r'[\u005D-\u007F]')

<short_bytes_char_no_double_quote> ::= (
      r'[\u0000-\u0009]'
    | r'[\u000B-\u000C]'
    | r'[\u000E-\u0021]'
    | r'[\u0023-\u005B]'
    | r'[\u005D-\u007F]')
```

```
<long_bytes> ::= (
      "'''" <long_bytes_item>* "'''"
    | '"""' <long_bytes_item>* '"""')

<long_bytes_item> ::= <long_bytes_char> | <bytes_escape_seq>
<long_bytes_char> ::= r'[\u0000-\u005B]' | r'[\u005D-\u007F]'
```

27.13 Numbers

Grammars can contain *numbers*, which are interpreted as bits (page 117). While the Fandango grammar supports arbitrary numbers, only the number literals `0` and `1` are supported (possibly with repetitions).

```
<number> ::= <integer> | <float_number> | <imag_number>
```

```
<integer> ::= <decimal_integer> | <oct_integer> | <hex_integer> | <bin_integer>
```

```
<decimal_integer> ::= <non_zero_digit> <digit>* | '0'+
<non_zero_digit> ::= r'[1-9]'
<digit> ::= r'[0-9]'
```

```
<oct_integer> ::= '0' r'[oO]' <oct_digit>+
<oct_digit> ::= r'[0-7]'

<hex_integer> ::= '0' r'[xX]' <hex_digit>+
<hex_digit> ::= r'[0-9a-fA-F]'

<bin_integer> ::= '0' r'[bB]' <bin_digit>+
<bin_digit> ::= r'[01]'
```

```
<float_number> ::= <point_float> | <exponent_float>
<point_float> ::= <int_part>? <fraction> | <int_part> '.'
<exponent_float> ::= ( <int_part> | <point_float> ) <exponent>
<int_part> ::= <digit>+
<fraction> ::= '.' <digit>+
<exponent> ::= r'[eE]' r'[+-]'? <digit>+

<imag_number> ::= (<float_number> | <int_part>) r'[jJ]'
```

27.14 Generators

A *generator* is an expression that is evaluated during fuzzing to produce a value, which is then *parsed* into the given nonterminal. See *the section on generators* (page 53) for details. It is added at the end of a production rule, separated by `:=`.

```
<generator> ::= <_> ':=' <_> <python_expression>
```

Future Fandango versions will also support invoking a generator as if it were a symbol.

```
<generator_call> ::= (<name>
    | <generator_call> '.' <name>
    | <generator_call> '[' <python_slices> ']'
    | <generator_call> <python_genexp>
    | <generator_call> '(' <python_arguments>? ')')
```

27.15 Constraints

A *constraint* is a Python expression that is to be satisfied during fuzzing and parsing. It is introduced by the keyword `where`.

Constraints typically contain symbols (page 143) (`<...>`); these are allowed wherever values are allowed. The constraint has to hold for all values of the given symbols.

Symbols in constraints have a `DerivationTree` type; see the *Derivation Tree Reference* (page 151) for details.

```
<constraint> ::= 'where' <_> <python_expression> <comment>? (';' | <newline>)
```

27.16 Selectors

Symbols in constraints can take the following special forms:

- `<A>.<B>`: The constraint has to hold for all elements `<B>` that are a direct child of `<A>`
- `<A>..<B>`: The constraint has to hold for all elements `<B>` that are a direct or indirect child of `<A>`

For details, see *the section on derivation trees* (page 151).

```
<selector> ::= (
      <selection>
    | <selector> '.' <selection>
    | <selector> '..' <selection>)
```

The `[...]` operator allows accessing individual children of a derivation tree:

```
<selection> ::= <base_selection> | <base_selection> '[' <python_slices> ']'
```

These operators can be combined and parenthesized:

```
<base_selection> ::= <nonterminal> | '(' <selector> ')'
```

27.17 Python Code

In a `.fan` file, anything that is neither a *grammar production rule* (page 138) nor a *constraint* (page 142) is interpreted as *Python code*, parsed as `<statement>` in the official Python grammar[36].

Also, in the above spec, any nonterminal in the form `<python_NAME>` (say, `<python_expression>`) refers to `<NAME>` (say, `<expression>`) in the official Python grammar[37].

For more details on Python syntax and semantics, consult the Python language reference[38].

27.18 The Full Spec

You can access the above spec `fandango.fan` for reference.

`fandango.fan` is sufficient for parsing `.fan` input without Python expressions or code:

```
$ echo '<start> ::= "a" | "b" | "c"' | fandango parse -f fandango.fan -o -
```

```
<start> ::= "a" | "b" | "c"
```

To complete the grammar, `fandango.fan` provides placeholders for included Python elements:

```
<python_statement> ::= 'pass' <newline>
<python_slices> ::= '0:1'
<python_arguments> ::= '1'
<python_expression> ::= '1' | <selector>
<python_genexp> ::= '[for' <_> <name> <_> 'in' <_> <name> ':' <_> <python_expression>
↪']'
```

[36] https://docs.python.org/3/reference/grammar.html
[37] https://docs.python.org/3/reference/grammar.html
[38] https://docs.python.org/3/reference/index.html

Hence, it is also possible to produce Fandango specs (with set Python code) using `fandango.fan`. Hence, Fandango can be fuzzed with itself:

```
$ fandango fuzz -f fandango.fan -n 1
```

```
  # 17
<ÎïKÍ7AÚgìådgxDSXû> ::= (<ÂuäMºqÞ> <o> | Z <0noÌ>){1} | <ùnäHÞmNahAD¼8bº8Và> <Õ_
 ↪MÐSÕìqßn4ÚÌ0çSr>  # +
;
```

Note that such generated files satisfy the Fandango syntax, but not its *semantics*. For instance, one would have to add extra constraints such that all used nonterminals are defined.

CHAPTER

TWENTYEIGHT

FANDANGO STANDARD LIBRARY

Fandango provides a set of predefined grammar symbols. Each symbol is defined as

- `<_SYMBOL>` (with the actual "official" definition), and
- `<SYMBOL>` (defined as `<_SYMBOL>` by default; can be overridden in individual specifications)

If you'd like to narrow the definition of, say, punctuation characters, you can redefine `<punctuation>` to your liking:

```
<punctuation> ::= '!' | '?' | ',' | '.' | ';' | ':'
```

The original definition of `<_punctuation>`, however, must not be changed, as other definitions may depend on it.

Warning

Symbols starting with an underscore must *not* be redefined.

28.1 Characters

A `<char>` represents any Unicode character, including newline.

```
<_char> ::= r'(.|\n)'
<char> ::= <_char>
```

28.2 Printable Characters

These symbols mimic the string constants from the Python `string` module[39]. Use `<digit>`, `<ascii_letter>`, `<whitespace>`, and more to your liking.

```
<_printable> ::= r'[0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!
↪\x22#$%&\x27()*+,-./:;<=>?@[\]^_`{|}~ \t\n\r\x0b\x0c]'
<printable> ::= <_printable>

<_whitespace> ::= r'[ \t\n\r\x0b\x0c]'
<whitespace> ::= <_whitespace>

<_digit> ::= r'[0-9]'
```

(continues on next page)

39 https://docs.python.org/3/library/string.html

(continued from previous page)

```
<digit> ::= <_digit>

<_hexdigit> ::= r'[0-9a-fA-F]'
<hexdigit> ::= <_hexdigit>

<_octdigit> ::= r'[0-7]'
<octdigit> ::= <_octdigit>

<_ascii_letter> ::= r'[a-zA-Z]'
<ascii_letter> ::= <_ascii_letter>

<_ascii_lowercase_letter> ::= r'[a-z]'
<ascii_lowercase_letter> ::= <_ascii_lowercase_letter>

<_ascii_uppercase_letter> ::= r'[A-Z]'
<ascii_uppercase_letter> ::= <_ascii_uppercase_letter>

<_punctuation> ::= r'[!\x22#$%&\x27()*+,-./:;<=>?@[\]^_`{|}~]'
<punctuation> ::= <_punctuation>

<_alphanum> ::= r'[a-zA-Z0-9]'
<alphanum> ::= <_alphanum>

<_alphanum> ::= r'[a-zA-Z0-9_]'
<alphanum> ::= <_alphanum>
```

28.3 Unicode Characters

A `<any_letter>` is any Unicode alphanumeric character, as well as the underscore (_).

An `<any_digit>` is any character in the Unicode character category `[Nd]`. This includes `[0-9]`, and also many other digit characters.

An `<any_whitespace>` is any Unicode whitespace character. This includes `[ \t\n\r\f\v]`, and also many other characters, for example the non-breaking spaces mandated by typography rules in many languages.

The symbols `<any_non_letter>`, `<any_non_digit>`, and `<any_non_whitespace>` match any character that is not in `<any_letter>`, `<any_digit>`, and `<any_whitespace>`, respectively.

```
<_any_letter> ::= r'\w'
<any_letter> ::= <_any_letter>

<_any_digit> ::= r'\d'
<any_digit> ::= <_any_digit>

<_any_whitespace> ::= r'\s'
<any_whitespace> ::= <_any_whitespace>

<_any_non_letter> ::= r'\W'
<any_non_letter> ::= <_any_non_letter>

<_any_non_digit> ::= r'\D'
<any_non_digit> ::= <_any_non_digit>
```

(continues on next page)

(continued from previous page)

```
<_any_non_whitespace> ::= r'\S'
<any_non_whitespace> ::= <_any_non_whitespace>
```

28.4 ASCII Characters

`<ascii_char>` expands into all 7-bit characters in the ASCII range 0x00-0x7F, printable and non-printable.

```
<_ascii_char> ::= rb'[\x00-\x7f]'
<ascii_char> ::= <_ascii_char>
```

28.5 ASCII Control Characters

`<ascii_control>` expands into any of the ASCII control characters 0x00-0x1F and 0x7F. We also provide ASCII codes such as `<ESC>`, `<LF>`, or `<NUL>`.

```
<_NUL> ::= b'\x00'
<NUL> ::= <_NUL>

<_SOH> ::= b'\x01'
<SOH> ::= <_SOH>

<_STX> ::= b'\x02'
<STX> ::= <_STX>

<_ETX> ::= b'\x03'
<ETX> ::= <_ETX>

<_EOT> ::= b'\x04'
<EOT> ::= <_EOT>

<_ENQ> ::= b'\x05'
<ENQ> ::= <_ENQ>

<_ACK> ::= b'\x06'
<ACK> ::= <_ACK>

<_BEL> ::= b'\x07'
<BEL> ::= <_BEL>

<_BS> ::= b'\x08'
<BS> ::= <_BS>

<_HT> ::= b'\x09'
<HT> ::= <_HT>

<_LF> ::= b'\x0a'
<LF> ::= <_LF>

<_VT> ::= b'\x0b'
<VT> ::= <_VT>
```

(continues on next page)

(continued from previous page)

```
<_FF> ::= b'\x0c'
<FF> ::= <_FF>

<_CR> ::= b'\x0d'
<CR> ::= <_CR>

<_SO> ::= b'\x0e'
<SO> ::= <_SO>

<_SI> ::= b'\x0f'
<SI> ::= <_SI>

<_DLE> ::= b'\x10'
<DLE> ::= <_DLE>

<_DC1> ::= b'\x11'
<DC1> ::= <_DC1>

<_DC2> ::= b'\x12'
<DC2> ::= <_DC2>

<_DC3> ::= b'\x13'
<DC3> ::= <_DC3>

<_DC4> ::= b'\x14'
<DC4> ::= <_DC4>

<_NAK> ::= b'\x15'
<NAK> ::= <_NAK>

<_SYN> ::= b'\x16'
<SYN> ::= <_SYN>

<_ETB> ::= b'\x17'
<ETB> ::= <_ETB>

<_CAN> ::= b'\x18'
<CAN> ::= <_CAN>

<_EM> ::= b'\x19'
<EM> ::= <_EM>

<_SUB> ::= b'\x1a'
<SUB> ::= <_SUB>

<_ESC> ::= b'\x1b'
<ESC> ::= <_ESC>

<_FS> ::= b'\x1c'
<FS> ::= <_FS>

<_GS> ::= b'\x1d'
<GS> ::= <_GS>

<_RS> ::= b'\x1e'
<RS> ::= <_RS>
```

(continues on next page)

(continued from previous page)

```
<_US> ::= b'\x1f'
<US> ::= <_US>

<_SP> ::= b'\x20'
<SP> ::= <_SP>

<_DEL> ::= b'\x7f'
<DEL> ::= <_DEL>

<_ascii_control> ::= <NUL> | <SOH> | <STX> | <ETX> | <EOT> | <ENQ> | <ACK> | <BEL>␣
↪| <BS> | <HT> | <LF> | <VT> | <FF> | <CR> | <SO> | <SI> | <DLE> | <DC1> | <DC2>␣
↪| <DC3> | <DC4> | <NAK> | <SYN> | <ETB> | <CAN> | <EM> | <SUB> | <ESC> | <FS> |
↪<GS> | <RS> | <US> | <SP> | <DEL>
<ascii_control> ::= <_ascii_control>
```

28.6 Bits

A `<bit>` represents a bit of 0 or 1. Use `<bit>{N}` to specify a sequence of `N` bits.

```
<_bit> ::= 0 | 1
<bit> ::= <_bit>
```

28.7 Bytes

A `<byte>` is any byte 0x00-0xFF. During parsing and production, it will always be interpreted as a single byte.

```
<_byte> ::= rb'[\x00-\xff]'
<byte> ::= <_byte>
```

28.8 UTF-8 characters

A `<utf8_char>` is a UTF-8 encoding of a character, occupying one (`<utf8_char1>`) to four (`<utf8_char4`) bytes.

```
<_utf8_char1> ::= rb'[\x00-\x7f]'
<utf8_char1> ::= <_utf8_char1>

<_utf8_continuation_byte> ::= rb'[\x80-\xbf]'
<utf8_continuation_byte> ::= <_utf8_continuation_byte>

<_utf8_char2> ::= rb'[\xc2-\xdf]' <utf8_continuation_byte>
<utf8_char2> ::= <_utf8_char2>

<_utf8_char3> ::= rb'[\xe0-\xef]' <utf8_continuation_byte>{2}
<utf8_char3> ::= <_utf8_char3>

<_utf8_char4> ::= rb'[\xf0-\xf5]' <utf8_continuation_byte>{3}
<utf8_char4> ::= <_utf8_char4>
```

(continues on next page)

(continued from previous page)

```
<_utf8_char> ::= <utf8_char1> | <utf8_char2> | <utf8_char3> | <utf8_char4>
<utf8_char> ::= <_utf8_char>
```

28.9 Binary Numbers

For binary representations of numbers, use symbols such as `<int8>` (8 bits) or `<int32>` (32 bits). Note that these symbols only specify the *length*; they do not cover signs, endianness, or byte ordering.

```
<_int8> ::= <byte>
<int8> ::= <_int8>

<_int16> ::= <byte><byte>
<int16> ::= <_int16>

<_int32> ::= <byte>{4}
<int32> ::= <_int32>

<_int64> ::= <byte>{8}
<int64> ::= <_int64>

<_float32> ::= <byte>{4}
<float32> ::= <_float32>

<_float64> ::= <byte>{8}
<float64> ::= <_float64>
```

28.10 Fandango Dancer

The `<fandango-dancer>` symbol is used to test UTF-8 compatibility.

```
<_fandango_dancer> ::= '⍰'
<fandango_dancer> ::= <_fandango_dancer>
```

CHAPTER

TWENTYNINE

DERIVATION TREE REFERENCE

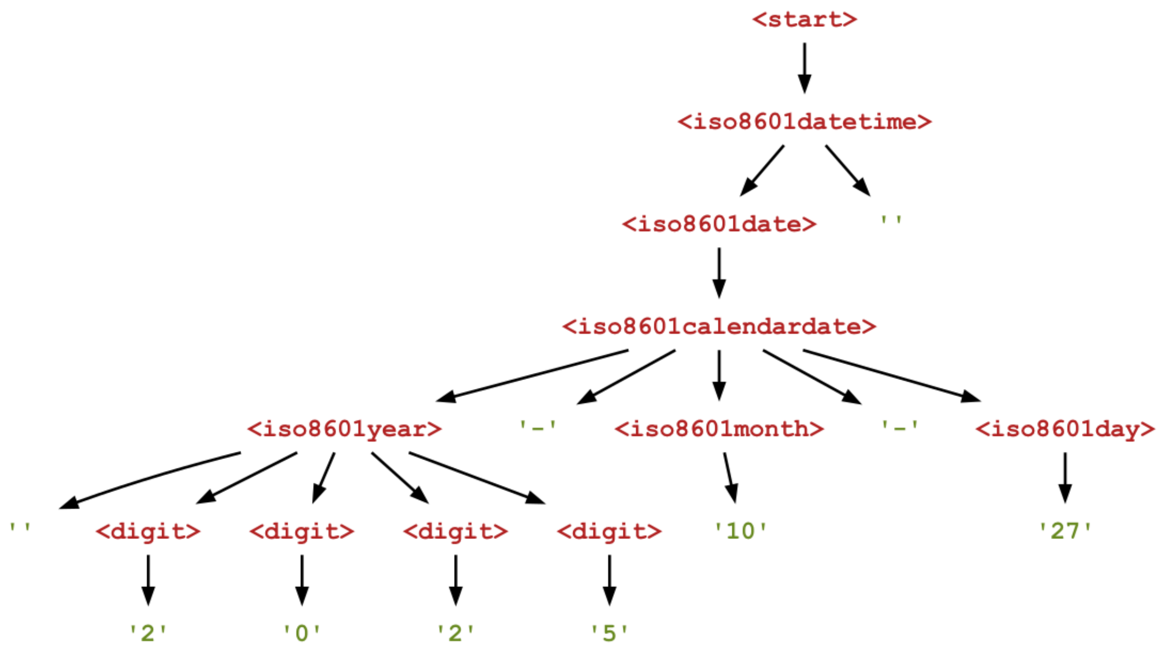
Fandango constraints make use of *symbols* (anything enclosed in `<...>`) to express desired or required properties of generated inputs. During evaluation, any `<SYMBOL>` returns a *derivation tree* representing the composition of the produced or parsed string.

This derivation tree has a type of `DerivationTree`, as do all subtrees. `DerivationTree` objects support *211 functions, methods, and operators* directly; after converting `DerivationTree` objects into standard Python types such as `str`, the entire Python ecosystem is available.

29.1 Derivation Tree Structure

Let's have a look at the structure of a derivation tree. Consider this derivation tree from the *ISO 8601 date and time grammar* (page 85):

Note that nonterminals can be empty strings, such as the second child of `<iso8601datetime>`.



The elements of the tree are designated as follows:

Node
: Any connected element of a tree.

Child
: An immediate descendant of a node. In the above tree, the `<start>` node has one child, `<iso8601datetime>`.

Descendant
: A child of a node, or a descendant of one of its children. `<iso8601month>` is a descendant of `<iso8601date>`. All nodes (except the root node) are descendants of the root node.

Parent
: The parent of a node N is the node that has N as a child. `<start>` is the parent of `<iso8601datetime>`.

Root
: The node without parent; typically `<start>`.

Terminal Symbol
: A node with at least one child.

Nonterminal Symbol
: A node without children.

Concatenating all terminal symbols (using `str(<SYMBOL>)` or `<SYMBOL>.value()`) in a derivation tree yields the string represented. For the above tree, this would be `2025-10-27`.

29.2 Evaluating Derivation Trees

Since standard Python functions do not accept derivation trees as arguments, one must first convert them into an acceptable type. The `value()` method plays a central role in this.

`<SYMBOL>.value() -> str | int | bytes`
: The method `<SYMBOL>.value()` *evaluates* `<SYMBOL>` and returns its value, which represents the concatenation of all descendants. The *type* of the return value is

  - A *Unicode string* (`str`) if all descendants are Unicode strings.
  - An *integer* (`int`) if all descendants are bits.
  - A *byte string* (`bytes`) in all other cases, notably if
    - any of the descendants of `<SYMBOL>` is a byte string, or
    - the descendants of `<SYMBOL>` contain bits *and* other elements.
  - `None` if `<SYMBOL>` expands into zero elements

29.2.1 Unicode Strings

Any derivation tree that is composed only from Unicode strings will evaluate into a Python *Unicode string* (`str`). The descendants of `<SYMBOL>` are concatenated as strings.

Example - in

```
<foo> ::= <bar> "bara"
<bar> ::= "bar"
```

`<foo>.value()` will be the Unicode string `"barbara"`.

This is the most common case.

29.2.2 Integers

Any derivation tree that is composed only from bits will evaluate into a Python *integer* (`int`). The descendants are concatenated as bits; the most significant bit comes first.

Example - in

```
<foo> ::= <bar> 0 1 1
<bar> ::= 0 1 0
```

`<foo>.value()` will be the integer `0b010011`, or 19.

29.2.3 Byte Strings

Any derivation tree where any of the descendants is a byte string, or where the descendants are bits *and* other elements, will evaluate into a Python *byte string* (`bytes`). The descendants are all concatenated as follows:

1. Any *bit sequence* B is converted into a *bytes string* with the most significant bit coming first.
2. Any *Unicode string* S is converted into a *bytes string* representing S in UTF-8 encoding.
3. The resulting byte strings are all concatenated.

Example 1 - in

```
<foo> ::= <bar> "foo"  # a Unicode string
<bar> ::= b"bar"       # a byte string
```

`<foo>.value()` will be the byte string `b"barfoo"`.

Warning

If you mix byte strings and Unicode strings a grammar, Fandango will issue a warning.

Example 2 - in

```
<foo> ::= <bar> b"foo"     # a byte string
<bar> ::= 1 1 1 1 1 1 1 1 # a bit string
```

`<foo>.value()` will be the byte string `b"foo\xff"`.

Warning

If you mix bits and Unicode strings in a grammar, Fandango will issue a warning.

29.3 General `DerivationTree` Functions

These functions are available for all `DerivationTree` objects, regardless of the type they evaluate into.

29.3.1 Converters

Invoking methods (`<SYMBOL>.METHOD()`), as well as operators (say, `<SYMBOL> + ...`), where `<SYMBOL>` is one of the operators, do not require conversion.

Since any `<SYMBOL>` has the type `DerivationTree`, one must convert it first into a standard Python type before passing it as argument to a standard Python function.

`str(<SYMBOL>) -> str`
Convert `<SYMBOL>` into a Unicode string. Byte strings in `<SYMBOL>` are converted using `latin-1` encoding.

`bytes(<SYMBOL>) -> bytes`
Convert `<SYMBOL>` into a byte string. Unicode strings in `<SYMBOL>` are converted using `utf-8` encoding.

`int(<SYMBOL>) -> int`
Convert `<SYMBOL>` into an integer, like the Python `int()` function. `<SYMBOL>` must be an `int`, or a Unicode string or byte string representing an integer literal.

`float(<SYMBOL>) -> float`
Convert `<SYMBOL>` into a floating-point number, like the Python `float()` function. `<SYMBOL>` must be an `int`, or a Unicode string or byte string representing a float literal.

`complex(<SYMBOL>) -> complex`
Convert `<SYMBOL>` into a complex number, like the Python `complex()` function. `<SYMBOL>` must be an `int`, or a Unicode string or byte string representing a float or complex literal.

`bool(<SYMBOL>) -> bool`
Convert `<SYMBOL>` into a truth value:

- If `<SYMBOL>` evaluates into an integer (because it represents bits), the value will be `True` if the integer is non-zero.
- If `<SYMBOL>` evaluates into a string (bytes or Unicode), the value will be `True` if the string is not empty.

Note that Python applies `bool()` conversion by default if a truth value is needed; hence, expressions like `<flag_1> and <flag_2>`, where both flags are bits, are allowed.

29.3.2 Node Attributes

`<SYMBOL>.sym() -> str | int | bytes`
The symbol of the node:

- for *nonterminals*, the symbol as a string (say, `"<SYMBOL>"`)
- for *terminals*, the value:
  - for Unicode strings, the value of the string (type `str`);
  - for bits, either `0` or `1` (type `int`)’

  – for bytes, the value of the byte string (type `bytes`).

`<SYMBOL>.is_terminal() -> bool`
True if `<SYMBOL>` is a terminal node.

`<SYMBOL>.is_nonterminal() -> bool`
True if `<SYMBOL>` is a nonterminal node.

`<SYMBOL>.is_regex() -> bool`
True if the (terminal) symbol of `<SYMBOL>` is a regular expression.

29.3.3 Accessing Children

`len(<SYMBOL>) -> int`
Return the number of children of `<SYMBOL>`.

Note

To access the length of the *string* represented by `<SYMBOL>`, use `len(str(<SYMBOL>))`.

`<SYMBOL>[n] -> DerivationTree`
Access the `n`th child of `<SYMBOL>`, as a `DerivationTree`. `<SYMBOL>[0]` is the first child; `<SYMBOL>[-1]` is the last child.

Note

To access the `n`th *character* of `<SYMBOL>`, use `str(<SYMBOL>)[n]`.

`<SYMBOL>[start:stop] -> DerivationTree`
Return a new `DerivationTree` which has the children `<SYMBOL>[start]` to `<SYMBOL>[stop-1]` as children. If `start` is omitted, children start from the beginning; if `stop` is omitted, children go up to the end, including the last one.

`<SYMBOL>.children() -> list[DerivationTree]`
Return a list containing all children of `<SYMBOL>`.

`<SYMBOL>.children_values() -> list[str | int | bytes]`
Return a list containing the values of all children of `<SYMBOL>`.

Note

Each element of the list can have a different type, depending on the type the `value()` method returns.

`<SYMBOL_1> in <SYMBOL_2>`
Return True if `<SYMBOL_1> == CHILD` for any of the children of `<SYMBOL_2>`.

`VALUE in <SYMBOL>`
Return True if `VALUE == CHILD.value()` for any of the children of `<SYMBOL>`.

29.3.4 Accessing Descendants

`<SYMBOL>.descendants() -> list[DerivationTree]`

Return a list containing all descendants of `<SYMBOL>`; that is, all children and their transitive children.

`<SYMBOL>.descendant_values() -> list[str | int | bytes]`

Return a list containing the values of all descendants of `<SYMBOL>`; that is, the values of all children and their transitive children.

Note

Each element of the list can have a different type, depending on the type the `value()` method returns.

29.3.5 Accessing Parents

`<SYMBOL>.parent() -> DerivationTree | None`

Return the parent of the current node, or `None` for the root node.

29.3.6 Accessing Sources

`<SYMBOL>.sources() -> list[DerivationTree]`

Return a list containing all sources of `<SYMBOL>`. Sources are symbols used in generator expressions out of which the value of `<SYMBOL>` was created; see *the section on data conversions* (page 109) for details.

29.3.7 Comparisons

`<SYMBOL_1> == <SYMBOL_2>`

Returns True if both trees have the same structure and all nodes have the same values.

`<SYMBOL> == VALUE`

Returns True if `<SYMBOL>.value() == VALUE`

`<SYMBOL_1> != <SYMBOL_2>`

Returns True if both trees have a different structure or any nodes have different values.

`<SYMBOL> == VALUE`

Returns True if `<SYMBOL>.value() == VALUE`.

`<SYMBOL_1> <|>|<=|>= <SYMBOL_2>`

Returns True if `<SYMBOL_1>.value() <|>|<=|>= <SYMBOL_2>.value()`.

`<SYMBOL> <|>|<=|>= VALUE`

Returns True if `<SYMBOL>.value() <|>|<=|>= VALUE`.

29.3.8 Debugging

See the *section on output formats* (page 98) for details on these representations.

`<SYMBOL>.to_bits() -> str`
: Return a bit representation (`0` and `1` characters) of `<SYMBOL>`.

`<SYMBOL>.to_grammar() -> str`
: Return a grammar-like representation of `<SYMBOL>`.

`<SYMBOL>.to_tree() -> str`
: Return a tree representation of `<SYMBOL>`, using `Tree(...)` constructors.

`repr(<SYMBOL>) -> str`
: Return the internal representation of `<SYMBOL>`, as a `DerivationTree` constructor that can be evaluated as a Python expression.

29.4 Type-Specific Functions

The bulk of available functions comes from the Python standard library.

29.4.1 Unicode Strings

For derivation trees that *evaluate* (page 152) into Unicode strings (`str`), all Python String methods[40] are available, such as `<SYMBOL>.startswith()`, `<SYMBOL>.endswith()`, `<SYMBOL>.strip()`, and more. The method is invoked on the `<SYMBOL>.value()` string value.

29.4.2 Integers

For derivation trees that *evaluate* (page 152) into integers (`int`), all Python numeric operators and functions[41] are available, including `+`, `-`, or `abs()`, as well as bitwise operators such as `<<`, `&`, `~`, etc. Symbols can be used on either side of an operator; the operator is applied on the `<SYMBOL>.value()` integer value.

In addition, the Python methods on integer types[42] can be used, such as `<SYMBOL>.to_bytes()` or `<SYMBOL>.bit_count()`. Methods are invoked on the `<SYMBOL>.value()` integer value.

29.4.3 Byte Strings

For derivation trees that *evaluate* (page 152) into byte strings (`bytes`), all Python bytes methods[43] are available, including `<SYMBOL>.decode()`, `<SYMBOL>.startswith()`, `<SYMBOL>.endswith()`, etc. The method is invoked on the `<SYMBOL>.value()` byte string value.

[40] https://docs.python.org/3/library/stdtypes.html#string-methods
[41] https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex
[42] https://docs.python.org/3/library/stdtypes.html#additional-methods-on-integer-types
[43] https://docs.python.org/3/library/stdtypes.html#bytes-and-bytearray-operations

CHAPTER

THIRTY

PYTHON API REFERENCE

Fandango provides a simple API for Python programs to

- load *a `.fan` specification with `Fandango()`* (page 159);
- produce outputs *from the spec with `fuzz()`* (page 160); and
- parse inputs *using the spec with `parse()`* (page 161).

Note

This API will be extended over time.

30.1 Installing the API

When you *install Fandango* (page 131), the Python Fandango API is installed as well.

30.2 The `Fandango` class

The Fandango API comes as a class, named `Fandango`. To use it, write

```
from fandango import Fandango
```

The `Fandango` constructor allows reading in a `.fan` specification, either from an (open) file, a string, or a list of strings or files.

```
class Fandango(fan_files: str | IO | List[str | IO],
    constraints: List[str] = None, *,
    start_symbol: Optional[str] = None,
    use_cache: bool = True,
    use_stdlib: bool = True,
    logging_level: Optional[int] = None)
```

Create a `Fandango` object.

- `fan_files` is the Fandango specification to load. This is either
  - a *string* holding a `.fan` specification; or
  - an (open) `.fan` *file*; or
  - a *list* of strings or files.

- `constraints`, if given, is a list of additional constraints (as strings).
- `start_symbol` is the start symbol to use (default: `<start>`).
- `use_cache` can be set to `False` to avoid loading the input from cache.
- `use_stdlib` can be set to `False` to avoid loading the *standard library* (page 145).
- `includes`: A list of directories to search for include files before the *Fandango spec locations* (page 167).
- `logging_level` controls the logging output. It can be set to any of the values in the Python logging module[44], such as `logging.DEBUG` or `logging.INFO`. Default is `logging.WARNING`.

Danger

Be aware that `.fan` files can contain Python code that is *executed when loaded*. This code can execute arbitrary commands.

Warning

Code in the `.fan` spec cannot access identifiers from the API-calling code or vice versa. However, as both are executed in the same Python interpreter, there is a risk that loaded `.fan` code may bypass these restrictions and gain access to the API-calling code.

Caution

Only load `.fan` files you trust.

`Fandango()` can raise a number of exceptions, including

- `FandangoSyntaxError` if the `.fan` input has syntax errors. The exception attributes `line`, `column`, and `msg` hold the line, column, and error message.
- `FandangoValueError` if the `.fan` input has consistency errors.

The exception class `FandangoError` is the superclass of these exceptions.

30.3 The `fuzz()` method

On a `Fandango` object, use the `fuzz()` method to produce outputs from the loaded specification.

```
fuzz(extra_constraints: Optional[List[str]] = None, **settings)
    -> List[DerivationTree])
```

Create outputs from the specification, as a list of *derivation trees* (page 151).

In the future, the set of available `settings` may change dependent on the chosen algorithm.

[44] https://docs.python.org/3/library/logging.html

- `extra_constraints`: if given, use this list of strings as additional constraints
- `settings`: pass extra values to control the fuzzer algorithm. These include
  - `population_size: int`: set the population size (default: 100).
  - `desired_solutions: int`: set the number of desired solutions.
  - `initial_population: List[Union[DerivationTree, str]]`: set the initial population.
  - `max_generations: int`: set the maximum number of generations (default: 500).
  - `warnings_are_errors: bool` can be set to True to raise an exception on warnings.
  - `best_effort: bool` can be set to True to return the population even if it does not satisfy the constraints.

The `fuzz()` method returns a list of *`DerivationTree` objects* (page 151). These are typically converted into Python data types (typically using `str()` or `bytes()`) to be used in standard Python functions.

`fuzz()` can raise a number of exceptions, including

- `FandangoFailedError` if the algorithm failed *and* `warnings_are_errors` is set.
- `FandangoValueError` if algorithm settings are invalid.
- `FandangoParseError` if a generator value could not be parsed.

The exception class `FandangoError` is the superclass of these exceptions.

30.4 The `parse()` method

On a `Fandango` object, use the `parse()` method to parse an input using the loaded specification.

```
parse(word: str | bytes | DerivationTree, *, prefix: bool = False, **settings)
    -> Generator[DerivationTree, None, None]
```

Parse a word; return a generator for *derivation trees* (page 151).

- `word`: the word to be parsed. This can be a string, a byte string, or a derivation tree.
- `prefix`: if True, allow incomplete inputs that form a prefix of the inputs accepted by the grammar.
- `settings`: additional settings for the parser.
  - There are no additional user-facing settings for the parser at this point.

The return value is a *generator* of derivation trees - if the `.fan` grammar is ambiguous, a single word may be parsed into different trees. To iterate over all trees parsed, use a construct such as

```
for tree in fan.parse(word):
    ...  # do something with the tree
```

`parse()` can raise exceptions, notably

- `FandangoParseError` if parsing fails. The attribute `position` of the exception indicates the position in `word` at which the syntax error was detected.[45]

Note

At this point, the `parse()` method does not check whether constraints are satisfied.

[45] Note that due to parser lookahead, the position may be slightly off the position of the actual error.

30.5 API Usage Examples

30.5.1 Fuzzing from a `.fan` Spec

Let us read and produce inputs from `persons-faker.fan` discussed in *the section on generators and fakers* (page 53):

```python
from fandango import Fandango

# Read in a .fan spec from a file
with open('persons-faker.fan') as persons:
    fan = Fandango(persons)

for tree in fan.fuzz(desired_solutions=10):
    print(str(tree))
```

```
John Clark,3541
Douglas Coffey,36
William Turner,34
Margaret Neal,45
Geoffrey Taylor,10
Judith Chase,92
Luke Hall,75
David Williams,81
Brent Black,45
Michael Smith,30
```

30.5.2 Fuzzing from a `.fan` String

We can also read a `.fan` spec from a string. This also demonstrates the usage of the `logging_level` parameter.

```python
from fandango import Fandango
import logging

# Read in a .fan spec from a string
spec = """
    <start> ::= ('a' | 'b' | 'c')+
    where str(<start>) != 'd'
"""

fan = Fandango(spec, logging_level=logging.INFO)
for tree in fan.fuzz(population_size=3):
    print(str(tree))
```

```
fandango:INFO: <string>: loading cached spec from /Users/zeller/Library/Caches/
↪Fandango/f94354f5584103bf2764367c84b3f839873d5d5a89704eb0baf8613220911a90.pickle
```

```
fandango:INFO: Redefining <start>
```

```
fandango:INFO: ---------- Initializing FANDANGO algorithm ----------
```

```
fandango:INFO: Generating initial population (size: 3)...
```

```
fandango:INFO: Initial population generated in 0.00 seconds
```

```
fandango:INFO: ---------- Starting evolution ----------
```

```
fandango:INFO: ---------- Evolution finished ----------
```

```
fandango:INFO: Perfect solutions found: (3)
```

```
fandango:INFO: Fitness of final population: 1.00
```

```
fandango:INFO: Time taken: 0.00 seconds
```

```
bcb
baacc
aca
```

30.5.3 Parsing an Input

This example illustrates usage of the `parse()` method.

```python
from fandango import Fandango

spec = """
    <start> ::= ('a' | 'b' | 'c')+
    where str(<start>) != 'd'
"""

fan = Fandango(spec)
word = 'abc'

for tree in fan.parse(word):
    print(f"tree = {repr(str(tree))}")
    print(tree.to_grammar())
```

```
tree = 'abc'
<start> ::= 'a' 'b' 'c'  # Position 0x0000 (0); 'abc'
```

Use the *`DerivationTree` functions* (page 151) to convert and traverse the resulting trees.

30.5.4 Parsing an Incomplete Input

This example illustrates how to parse a prefix (`'ab'`) for a grammar that expects a final `d` letter.

```python
from fandango import Fandango

spec = """
    <start> ::= ('a' | 'b' | 'c')+ 'd'
    where str(<start>) != 'd'
"""

fan = Fandango(spec)
```

(continues on next page)

(continued from previous page)

```
word = 'ab'

for tree in fan.parse(word, prefix=True):
    print(f"tree = {repr(str(tree))}")
    print(tree.to_grammar())
```

```
tree = 'ab'
<start> ::= 'a' 'b'  # Position 0x0000 (0); 'ab'
```

Without `prefix=True`, parsing would fail:

```
from fandango import Fandango

spec = """
    <start> ::= ('a' | 'b' | 'c')+ 'd'
    where str(<start>) != 'd'
"""

fan = Fandango(spec)
word = 'ab'

fan.parse(word)
```

```
Traceback (most recent call last):

  File ~/.virtualenvs/python3.12/lib/python3.12/site-packages/IPython/core/
↪interactiveshell.py:3667 in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)

  Cell In[8], line 11
    fan.parse(word)

  File ~/Projects/Fandango!/src/fandango/__init__.py:230 in parse
    raise FandangoParseError(position=position)

  File <string>
FandangoParseError: Parse error at position 3
```

30.5.5 Handling Parsing Errors

This example illustrates handling of `parse()` errors.

```
from fandango import Fandango, FandangoParseError

spec = """
    <start> ::= ('a' | 'b' | 'c')+
    where str(<start>) != 'd'
"""

fan = Fandango(spec)
invalid_word = 'abcdef'

try:
    fan.parse(invalid_word)
```

(continues on next page)

(continued from previous page)

```python
except FandangoParseError as exc:
    error_position = exc.position
    print("Syntax error at", repr(invalid_word[error_position]))
```

```
Syntax error at 'e'
```

CHAPTER

THIRTYONE

FANDANGO INCLUDE REFERENCE

31.1 Where Does Fandango Look for Included Specs?

In a Fandango spec, *the `include()` function* (page 125) searches for Fandango files in a number of locations. The actual rules for where Fandango searches are complex, versatile, and adhere to common standards. Specifically, when including a `.fan` file using `include()`, Fandango searches for `.fan` files in the following locations, in this order:

1. In any directory `DIR` explicitly specified by `-I DIR` or `--include-dir DIR`.
2. In any directory specified by the `$FANDANGO_PATH` environment variable, if set. This variable is a colon-separated list of directories, e.g. `$HOME/my_cool_fan_specs:/Volumes/Fandango:/opt/local/fandango-1.27` which are searched from left to right.
3. In the directory of the *including file*. This is the most common usage. If the including file has no file name (say, because it is a stream), the current directory is used.
4. In the directory `$HOME/.local/share/fandango`. You can control this location by setting the `$XDG_DATA_HOME` environment variable; see the XDG Base Directory Specification[46]. On a Mac, the location `$HOME/Library/Fandango` is searched first.
5. In one of the system directories `/usr/local/share/fandango` or `/usr/share/fandango`. You can control these locations by setting the `$XDG_DATA_DIRS` environment variable; see the XDG Base Directory Specification[47]. On a Mac, the location `/Library/Fandango` is searched first.
6. If the file to be included cannot be found in any of these locations, a `FileNotFoundError` is raised.

And then, there are two extra rules:

- If the included file has an *absolute* path (say, `/usr/local/specs/spec.fan`), only this absolute path will be used.
- If the included file has a *relative* path (say, `../spec.fan`), then this relative path will be applied to the locations above. This is mostly useful when you have a path relative to the including file.

Tip

We recommend to store included Fandango specs either

- in the directory where the *including* specs are, or
- in `$HOME/.local/share/fandango` (or `$HOME/Library/Fandango` on a Mac).

[46] https://specifications.freedesktop.org/basedir-spec/latest/
[47] https://specifications.freedesktop.org/basedir-spec/latest/

Finally, keep in mind that `include()` is a Python function, so in principle, your spec can compute the file name and location in any way you like. The simplest solution, however, is to use one of the "standard" locations as listed above.