The Earley Lark parses more: from structured text to data classes

Dan Jones

Hey, I’m Dan

Just your average coffee enjoyer who writes code

- From the UK
- Living in NL
- Studied MEng Computer Science at University of Bristol
- Enjoys: ☕ 🧗 🍻 </>

An Advent of Code esque file

```
6
10 10
23 24
1 2
5 3
123 4
1 1345
```

Quick and dirty

`str.split` to the rescure

- Okay for simple files
- Hard to read
- Harder to debug as complexity grows

```python
def parse_input(path: Path) -> list[tuple[int, int]]:
  content = path.read_text().splitlines()
  num_points = int(content[0])
  results: list[tuple[int, int]] = []
  for line in content[1:]:
    x, y = line.split(" ", maxsplit=1)
    results.append((int(x), int(y))
  return resutls
```

Regex is a bit nicer…

- More robust
- Complexity explodes with file complexity
  - `if` … `if` … `else` … `if` …
- What was the format?

```python
import re

def parse_input(path: Path) → list[tuple[int, int]]:
  count_pattern = re.Pattern(r"^(?P<count>\d+)$")
  coord_pattern = re.Pattern(r"^(?P<x>\d+) (?P<y>\d+)$")

  content = path.read_text().splitlines()
  count_match = count_pattern.match(content[0])
  if not count_match:
    raise ParseError("No valid count")

  num_points = int(count_match.group("count"))

  results: list[tuple[int, int]] = []
  for line in content[1:]:
    match = coord_pattern.match(line)
    if not match:
      raise ParseError("Could not parse coord")
    coord = (int(match.group("x")), int(match.group("y")))
    results.append(coord)
  return resutls
```

A better way?

Lark aims to make parsing and aims to:

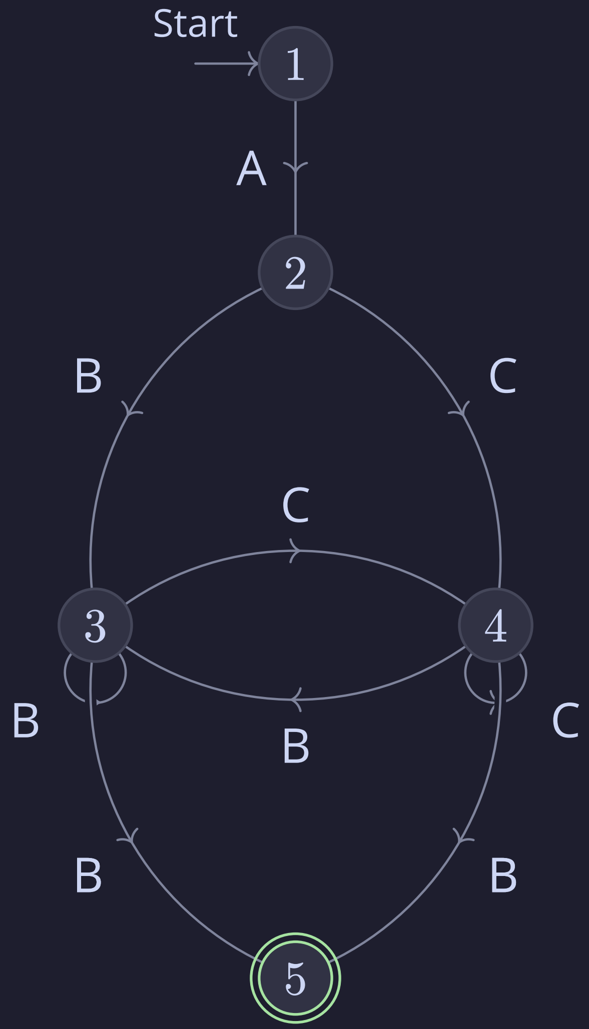
1. be readable
2. be clean and simple
3. be usable

Is the input valid?

A step back to university

Regular Languages

- Parsed via a *deterministic finite automata*
- A REGEX encodes a Regular Language



Regex can't parse everything

Can you parse `A{n}B{n}`?[1]

[1] we want to parse a string with equal number A's and B's

Introducing CFGs

- Superset of regular languages
- Written in extended Backus-Naur form (EBNF)

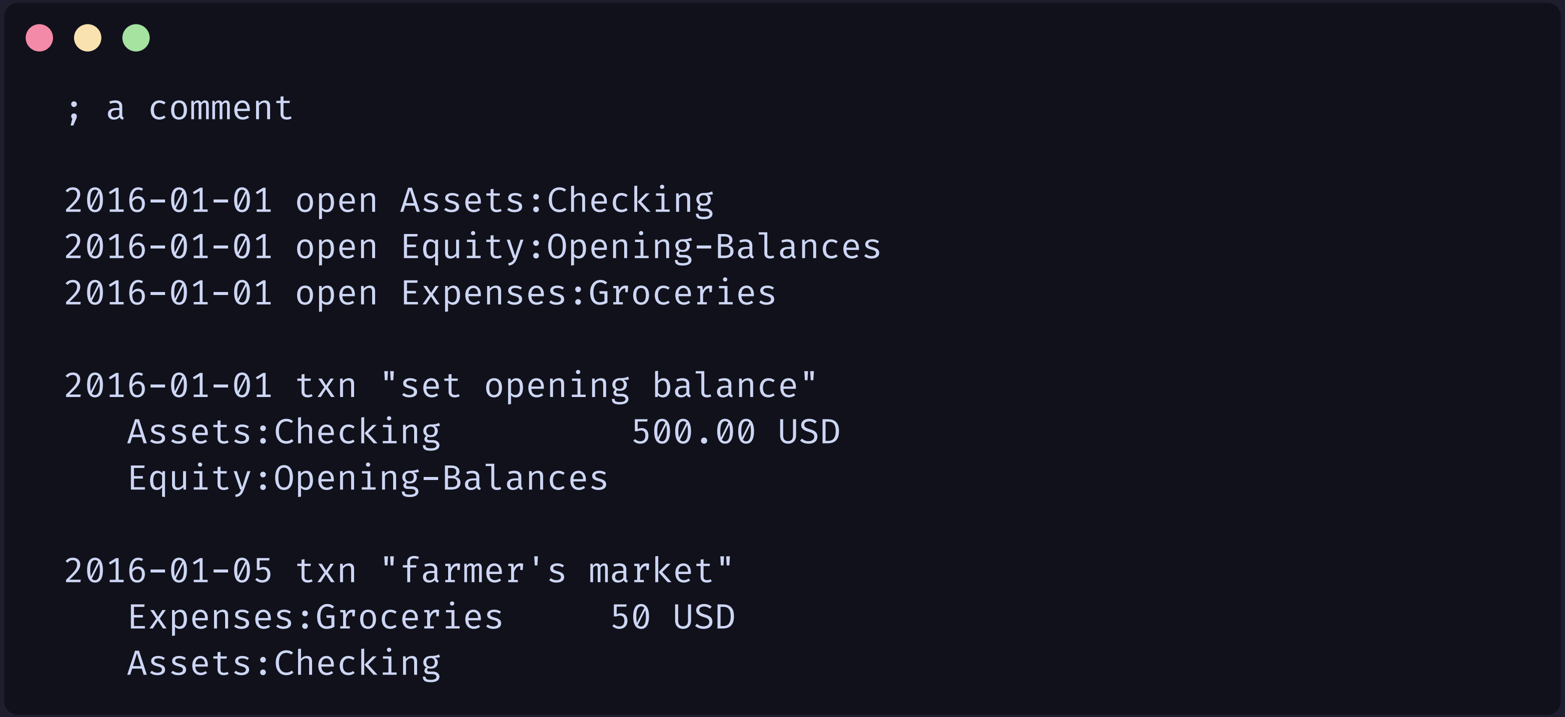
$$S \mapsto aRb$$
$$R \mapsto aRb \mid \varepsilon$$

To parse `a{n}b{n}`:

1. $aabb$
2. Apply S: $\cancel{a}ab\cancel{b} \mapsto ab$
3. Apply R: $\cancel{a}\cancel{b} \mapsto \varepsilon$
4. After R: $\cancel{\varepsilon} \mapsto$ validated ✓

Parsing with Lark

Plain Text Accounting

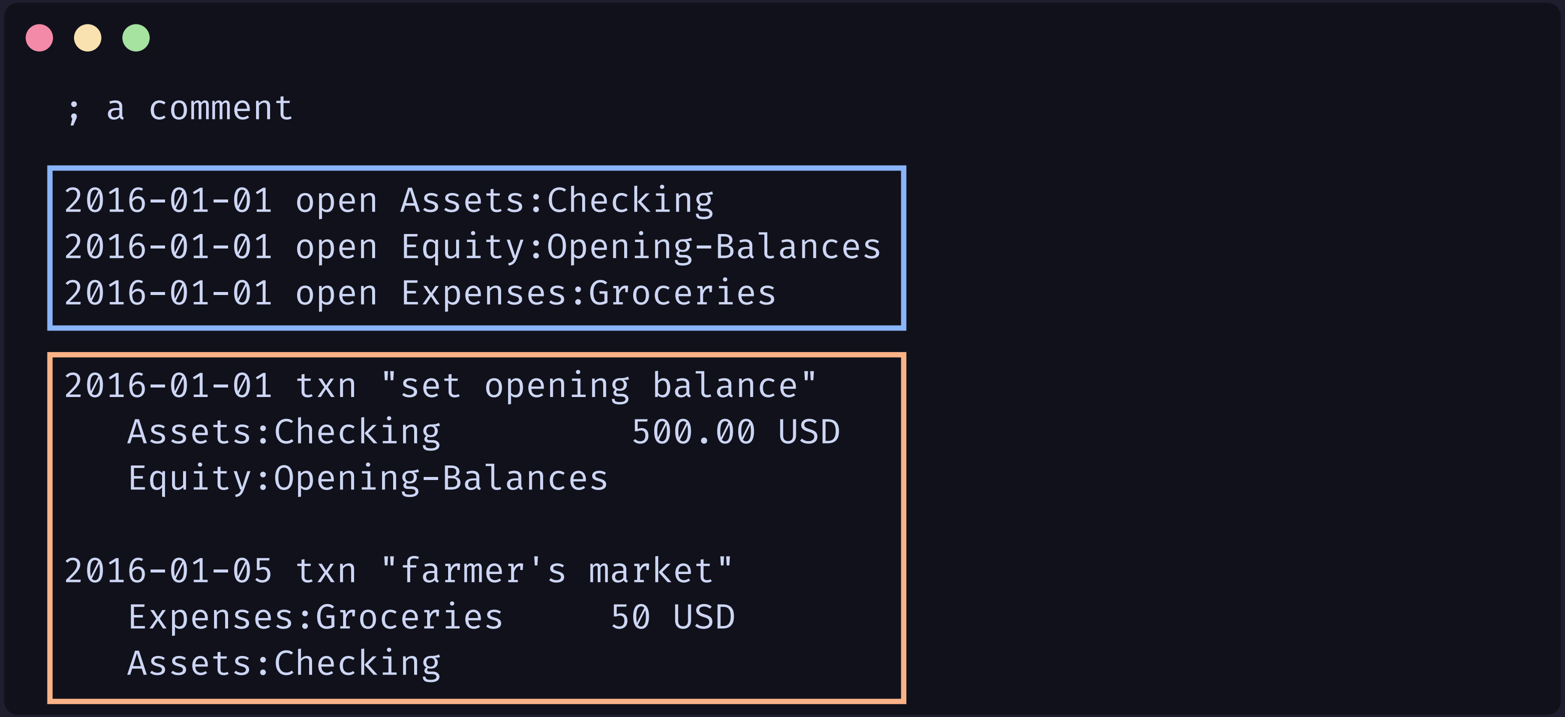


```
; a comment

2016-01-01 open Assets:Checking
2016-01-01 open Equity:Opening-Balances
2016-01-01 open Expenses:Groceries

2016-01-01 txn "set opening balance"
   Assets:Checking         500.00 USD
   Equity:Opening-Balances

2016-01-05 txn "farmer's market"
   Expenses:Groceries     50 USD
   Assets:Checking
```

Plain Text Accounting

```
; a comment

2016-01-01 open Assets:Checking
2016-01-01 open Equity:Opening-Balances
2016-01-01 open Expenses:Groceries

2016-01-01 txn "set opening balance"
   Assets:Checking         500.00 USD
   Equity:Opening-Balances

2016-01-05 txn "farmer's market"
   Expenses:Groceries     50 USD
   Assets:Checking
```

Plain Text Accounting

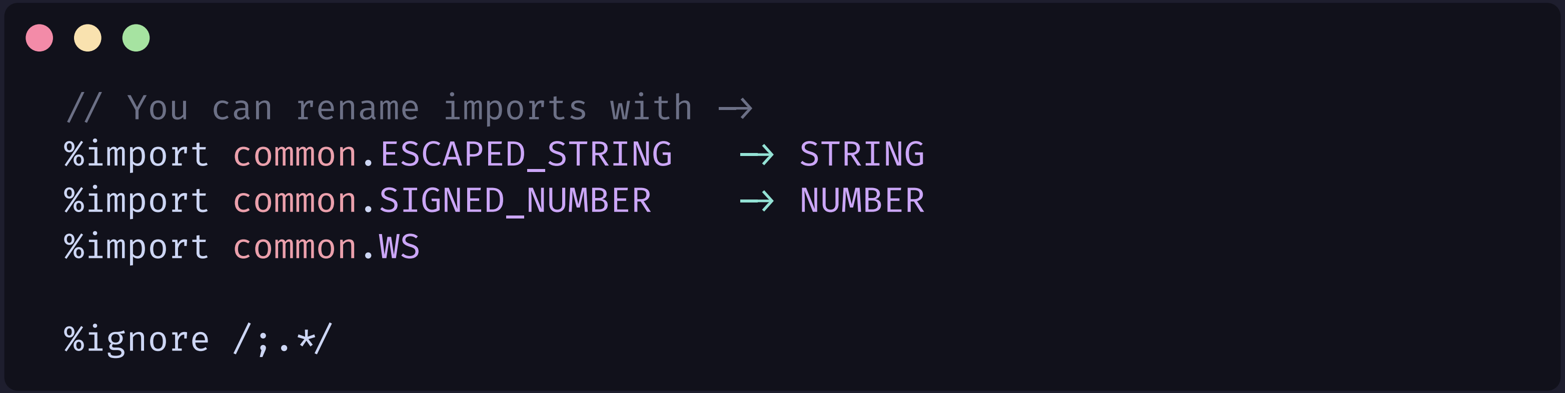


```
; a comment

2016-01-01 open Assets:Checking
2016-01-01 open Equity:Opening-Balances
2016-01-01 open Expenses:Groceries

2016-01-01 txn "set opening balance"
   Assets:Checking          500.00 USD
   Equity:Opening-Balances

2016-01-05 txn "farmer's market"
   Expenses:Groceries     50 USD
   Assets:Checking
```

Creating the grammar: drop comments

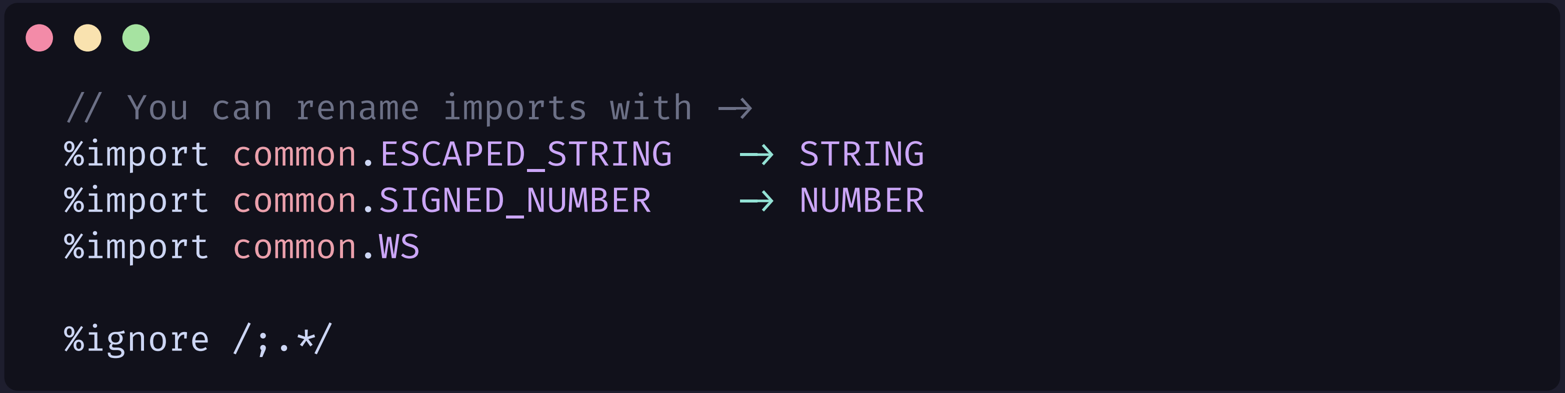
```
// This is a comments in Lark

%ignore /;.*/   // ← Look a REGEX
```

Creating the grammar: import standard terminals



```
// You can rename imports with ->
%import common.ESCAPED_STRING   -> STRING
%import common.SIGNED_NUMBER    -> NUMBER
%import common.WS

%ignore /;.*/
```

Creating the grammar: import standard terminals



```
// You can rename imports with →
%import common.ESCAPED_STRING   → STRING
%import common.SIGNED_NUMBER    → NUMBER
%import common.WS

%ignore /;.*/
```

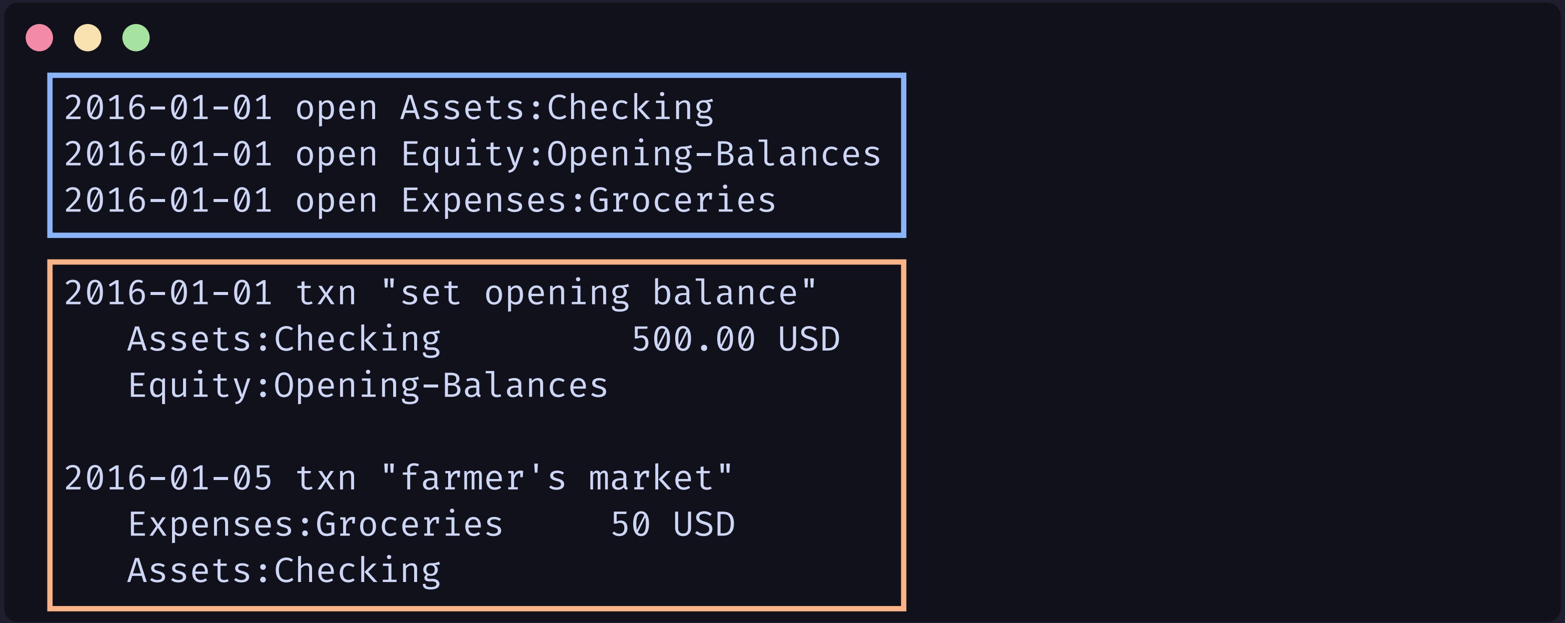
Creating the grammar: our terminals

Where everything ends at

```
// Note: terminals are UPPER case

ACCOUNT_NAME: /\w+:\w+/
DATE: /\d{4}[-/.]\d{2}[-/.]\d{2}/  // YYYY-MM-DD, YYYY.MM.DD, YYYY/MM/DD
PUT_CALL: "put" | "call"

%import common.ESCAPED_STRING   → QUOTE_STRING
%import common.SIGNED_NUMBER    → NUMBER
%import common.WS
%import common.NL

%ignore /;.*/
```

Plain Text Accounting: start rules

Remember two sections



```
2016-01-01 open Assets:Checking
2016-01-01 open Equity:Opening-Balances
2016-01-01 open Expenses:Groceries
```

```
2016-01-01 txn "set opening balance"
   Assets:Checking         500.00 USD
   Equity:Opening-Balances

2016-01-05 txn "farmer's market"
   Expenses:Groceries     50 USD
   Assets:Checking
```

Creating the grammar: starting rules

```
root: account* WS* transaction*

account: // TODO

transition: // TODO

ACCOUNT_NAME: /\w+:\w+/
ASSET_NAME: /[A-Z]+/
DATE: /\d{4}[-/.]\d{2}[-/.]\d{2}/  // YYYY-MM-DD, YYYY.MM.DD, YYYY/MM/DD

// ... imports and ignores
```

Creating the grammar: the account line

```
root: account* WS* transaction*

account: DATE "open" ACCOUNT_NAME NL

transition: // TODO

ACCOUNT_NAME: /\w+:\w+/
DATE: /\d{4}[-/.]\d{2}[-/.]\d{2}/  // YYYY-MM-DD, YYYY.MM.DD, YYYY/MM/DD

// ... imports and ignores
```

Creating the grammar: transactions

```
root: account* WS* transaction*

account: DATE "open" ACCOUNT_NAME NL

transition: transaction_start full_posting+ final_posting
transaction_start: DATE "txn" QUOTE_STRING
full_posting: ACCOUNT_NAME amount
final_posting: ACCOUNT_NAME [amount]

amount: NUMBER ASSET_NAME

ACCOUNT_NAME: /\w+:\w+/
DATE: /\d{4}[-/.]\d{2}[-/.]\d{2}/  // YYYY-MM-DD, YYYY.MM.DD, YYYY/MM/DD

// ... imports and ignores
```

Parsing a file with our grammar

```python
from lark import Lark

def parse_ledger(ledger_file: Path) -> Lark.Tree:
  parser = Lark(Path("./grammar.lark").read_text())
  return parser.parse(ledger_file.read_text())
```

Parsing a file with our grammar

```python
from lark import Lark

def parse_ledger(ledger_file: Path) → Lark.Tree:
  parser = Lark(Path("./grammar.lark").read_text())
  return parser.parse(ledger_file.read_text())
```

🚀 we've parsed and validated the file!

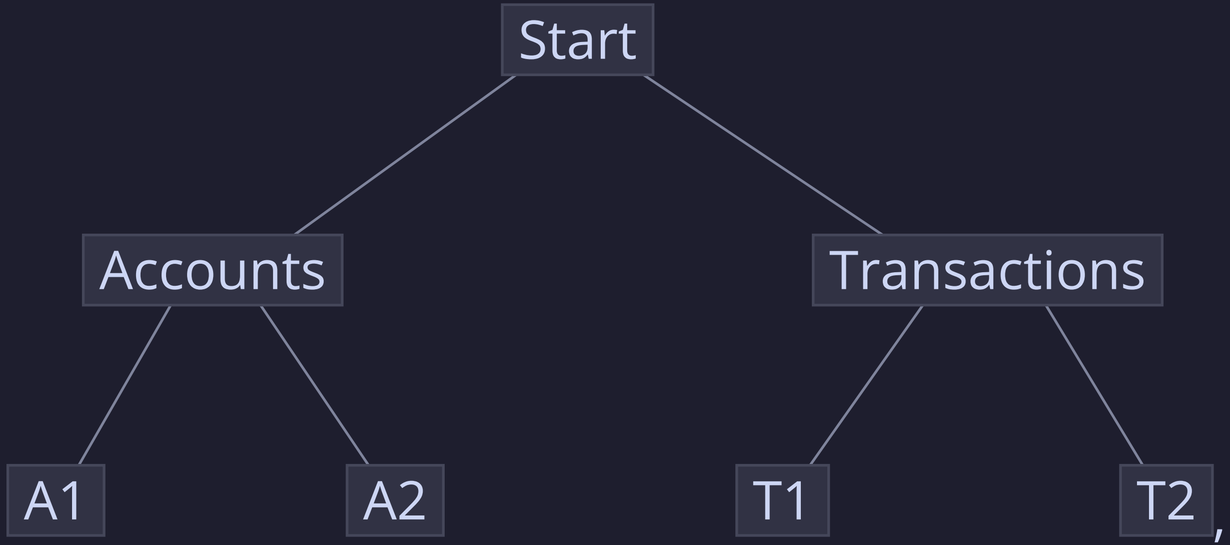
To dataclasses

Define some dataclasses

```python
from dataclasses import dataclass
from datetime import date
from typing import NewType

AccountName = NewType("AccountName", str)
AssetName = NewType("AssetName", str)


@dataclass()
class Account:
  open_date: date
  name: AccountName
```

```python
@dataclass()
class Posting:
  account: AccountName
  amount: float
  asset: AssetName


@dataclass()
class Transaction:
  dated: date
  postings: list[Posting]
```

Lark gives us a tree



Raise to dataclasses via transfomrers

```python
@v_args(inline=True)
class LedgerTransfomrer(Transformer):
  def start(
    self, accounts: list[Account],
    transactions: list[Transactions]
  ): ...

  def account(
    self, dated: date, name: AcocuntName
  ) → Account:
    return Account(dated=dated, name=name)

  def ACCOUNT_NAME(self, node) → AccountName:
    return AccountName(node)

  def DATE(self, node) → date:
    return date.parse(node)

LedgerTransformer().transform(tree)
```

```
root: account* WS* transaction*

account: DATE "open" ACCOUNT_NAME NL

ACCOUNT_NAME: /\w+:\w+/
DATE: /\d{4}[-/.]\d{2}[-/.]\d{2}/
```

Lark tips

- use `[val]` over `val?` in grammars
  - `[val]` gives explicit `None`
- `print(tree)` will show you the parsed tree
- `@v_args(inline=True)` parses all items explicitly
- Pair with Pydantic for even easier type coercion and validation

Summary

- > Writing code is communicating to your future self and other developers
  >
  > — Miriam Forner @ EuroPython 2024
- Lark can make parsing structured text easy[2]
- Seperating transforming and grammars is nice
- Lark is fun

[2]probably

Questions?

Dan Jones

danjones.dev

danjones1618

/in/danjones1618