Beancount Language Syntax

Martin Blais, Updated: April 2016

http://furius.ca/beancount/doc/syntax

Introduction

This is a user's manual to the language of Beancount, the command-line double-entry bookkeeping system. Beancount defines a computer language that allows you to enter financial transactions in a text file and extract various reports from it. It is a generic counting tool that works with multiple currencies, commodities held at cost (e.g., stocks), and even allows you to track unusual things, like vacation hours, air miles and rewards points, and anything else you might want to count, even beans.

This document provides an introduction to Beancount's syntax and some of the technical details needed for one to understand how it carries out its calculations. This document does *not* provide an introduction to the double-entry method, a motivation, nor examples and guidelines for entering transactions in your input file, nor how to run the tools. These subjects have their own dedicated documents, and it is recommended that you have had a look at those before diving into this user's manual. This manual covers the technical details for using Beancount.

Syntax Overview

Directives

Beancount is a declarative language. The input consists of a text file containing mainly a list of **directives**, or **entries** (we use these terms interchangeably in the code and documentation); there is also syntax for defining various **options**. Each directive begins with an associated *date*, which determines the point in time at which the directive will apply, and its *type*, which defines which kind of event this directive represents. All the directives begin with a syntax that looks like this:

YYYY-MM-DD <type> …

where YYYY is the year, MM is the numerical month, and DD the numerical date. All digits are required, for example, the 7th of May 2007 should be "2007-05-07", including its zeros. Beancount supports the international ISO 8601 standard format for dates, with dashes (e.g., "2014-02-03"), or the same ordering with slashes (e.g., "2014/02/03").

Here are some example directives, just to give you an idea of the aesthetics:

```
2014-02-03 open Assets:US:BofA:Checking

2014-04-10 note Assets:US:BofA:Checking "Called to confirm wire transfer."

2014-05-02 balance Assets:US:BofA:Checking   154.20 USD
```

The end product of a parsed input file is a simple list of these entries, in a data structure. All operations in Beancount are performed on these entries.

Each particular directive type is documented in a section below.

Ordering of Directives

The order of declaration of the directives is not important. In fact, the entries are re-sorted chronologically after parsing and before being processed. This is an important feature of the language, because it makes it possible for you to organize your input file any way you like without having to worry about affecting the meaning of the directives.

Except for transactions, each directive is assumed to occur at the *beginning* of each day. For example, you could declare an account being opened on the same day as its first transaction:

```
2014-02-03 open Assets:US:BofA:Checking

2014-02-03 * "Initial deposit"
  Assets:US:BofA:Checking         100 USD
```

```
      Assets:Cash                    -100 USD
```

However, if you hypothetically closed that account immediately, you could not declare it closed on the same day, you would have to fudge the date forward by declaring the close on 2/4:

```
2014-02-04 close Assets:US:BofA:Checking
```

This also explains why balance assertions are verified before any transactions that occur on the same date. This is for consistency.

Accounts

Beancount accumulates commodities in accounts. The names of these accounts do not have to be declared before being used in the file, they are recognized as "accounts" by virtue of their syntax alone[1]. An account name is a colon-separated list of capitalized words which begin with a letter, and whose first word must be one of five account types:

Assets
Liabilities
Equity
Income
Expenses

Each component of the account names begin with a capital letter or a number and are followed by letters, numbers or dash (-) characters. All other characters are disallowed.

Here are some realistic example account names:

```
Assets:US:BofA:Checking
Liabilities:CA:RBC:CreditCard
Equity:Retained-Earnings
Income:US:Acme:Salary
Expenses:Food:Groceries
```

The set of all names of accounts seen in an input file implicitly define a hierarchy of accounts (sometimes called a **chart-of-accounts**), similarly to how files are organized in a file system. For example, the following account names:

```
Assets:US:BofA:Checking
Assets:US:BofA:Savings
Assets:US:Vanguard:Cash
Assets:US:Vanguard:RGAGX
Assets:Receivables
```

implicitly declare a tree of accounts that looks like this:

```
`-- Assets
    |-- Receivables
    `-- US
        |-- BofA
        |   |-- Checking
        |   `-- Savings
```

[1] Note that there exists an "Open" directive that is used to provide the start date of each account. That can be located anywhere in the file, it does not have to appear in the file somewhere before you use an account name. You can just start using account names in transactions right away, though all account names that receive postings to them will eventually have to have a corresponding Open directive with a date that precedes all transactions posted to the account in the input file.

```
                      `-- Vanguard
                          |-- Cash
                          `-- RGAGX
```

We would say that “`Assets:US:BofA`” is the parent account of “`Assets:US:BofA:Checking`”, and that the latter is a child account of the former.

Commodities / Currencies

Accounts contain **currencies**, which we sometimes also call **commodities** (we use both terms interchangeably). Like account names, currency names are recognized by their syntax, though, unlike account names, they need not be declared before being used). The syntax for a currency is a word all in capital letters, like these:

```
USD
CAD
EUR
MSFT
IBM
AIRMILE
```

(Technically, a currency name may be up to 24 characters long, and it must start with a capital letter, must end with with a capital letter or number, and its other characters must only be capital letters, numbers, or punctuation limited to these characters: “'._-” (apostrophe, period, underscore, dash.) The first three might evoke real world currencies to you (US dollars, Canadian dollars, Euros); the next two, stock ticker names (Microsoft and IBM). And the last: rewards points (airmiles). Beancount knows of no such thing; from its perspective all of these instruments are treated similarly. There is no built-in notion of any previously existing currency. These currency names are just names of “things” that can be put in accounts and accumulated in inventories associated with these accounts.

There is something elegant about the fact that there is no “special” currency unit, that all commodities are treated equally the same: Beancount is inherently a multi-currency system. You will appreciate this if, like many of us, you are an expat and your life is divided between two or three continents. You can handle an international ledger of accounts without any problems.

And your use of currencies can get quite creative: you can create a currency for your home, for example (e.g. `MYLOFT`), a currency to count accumulated vacation hours (`VACHR`), or a currency to count potential contributions to your retirement accounts allowed annually (`IRAUSD`). You can actually solve many problems this way. The cookbook describes many such concrete examples.

Beancount does not support the dollar sign syntax, e.g., “$120.00”. You should always use names for currencies in your input file. This makes the input more regular and is a design choice. For monetary units, I suggest that you use the standard ISO 4217 currency code as a guideline; these quickly become familiar. However, as detailed above, you may include some other characters in currency names, like underscores (_), dashes (-), periods (.), or apostrophes (‘), but no spaces.

Finally, you will notice that there exists a “`commodity`” directive that can be used to declare currencies. It is entirely optional: currencies come into being as you use them. The purpose of the directive is simply to attach metadata to it.

Strings

Whenever we need to insert some free text as part of an entry, it should be surrounded by double-quotes. This applies to the payee and narration fields, mainly; basically anything that's not a date, a number, a currency, an account name.

Strings may be split over multiple lines. (Strings with multiple lines will include their newline characters and those need to be handled accordingly when rendering.)

Comments

The Beancount input file isn't intended to contain only your directives: you can be liberal in placing comments and headers in it to organize your file. Any text on a line after the character ";" is ignored, text like this:

```
; I paid and left the taxi, forgot to take change, it was cold.
2015-01-01 * "Taxi home from concert in Brooklyn"
  Assets:Cash      -20 USD  ; inline comment
  Expenses:Taxi
```

You can use one or more ";" characters if you like. Prepend on all lines if you want to enter a larger comment text. If you prefer to have the comment text parsed in and rendered in your journals, see the Note directive elsewhere in this document.

Any line that does not begin as a valid Beancount syntax directive (e.g. with a date) is silently ignored. This way you can insert markup to organize your file for various outline modes, such as org-mode in Emacs. For example, you could organize your input file by institution like this and fold & unfold each of the sections independently,:

```
* Banking
** Bank of America

2003-01-05 open Assets:US:BofA:Checking
2003-01-05 open Assets:US:BofA:Savings

;; Transactions follow …

** TD Bank

2006-03-15 open Assets:US:TD:Cash

;; More transactions follow …
```

The unmatching lines are simply ignored.

Note to visiting Ledger users: In Ledger, ";" is used both for marking comments and for attaching "Ledger tags" (Beancount metadata) to postings. This is not the case in Beancount. In Beancount comments are always just comments. Metadata has its own separate syntax.

Directives

For a quick reference & overview of directive syntax, please consult the Syntax Cheat Sheet.

Open

All accounts need to be declared "open" in order to accept amounts posted to them. You do this by writing a directive that looks like this:

```
2014-05-01 open Liabilities:CreditCard:CapitalOne     USD
```

The general format of the Open directive is:

*YYYY-MM-DD **open** Account [ConstraintCurrency,...] ["BookingMethod"]*

The comma-separated optional list of constraint currencies enforces that all changes posted to this account are in units of one of the declared currencies. Specifying a currency constraint is recommended: the more constraints you provide Beancount with, the less likely you will be to make data entry mistakes because if you do, it will warn you about it.

Each account should be declared "opened" at a particular date that precedes (or is the same as) the date of the first transaction that posts an amount to that account. Just to be clear: an Open directive does not have to appear before transactions *in the file*, but rather, the *date* of the Open directive must precede the date of postings to that account. The order of declarations in the file is not important. So for example, this is a legal input file:

```
2014-05-05 * "Using my new credit card"
  Liabilities:CreditCard:CapitalOne         -37.45 USD
  Expenses:Restaurant

2014-05-01 open Liabilities:CreditCard:CapitalOne     USD
1990-01-01 open Expenses:Restaurant
```

Another optional declaration for opening accounts is the "booking method", which is the algorithm that will be invoked if a reducing lot leads to an ambiguous choice of matching lots from the inventory (0, 2 or more lots match). Currently, the possible values it can take are:

- **STRICT**: The lot specification has to match exactly one lot. This is the default method. If this booking method is invoked, it will simply raise an error. This ensures that your input file explicitly selects all matching lots.
- **NONE**: No lot matching is performed. Lots of any price will be accepted. A mix of positive and negative numbers of lots for the same currency is allowed. (This is similar to how Ledger treats matching… it ignores it.)

Close

Similarly to the Open directive, there is a Close directive that can be used to tell Beancount that an account has become inactive, for example:

```
; Closing credit card after fraud was detected.
2016-11-28 close Liabilities:CreditCard:CapitalOne
```

The general format of the Close directive is:

*YYYY-MM-DD **close** Account*

This directive is used in a couple of ways:

- An error message is raised if you post amounts to that account after its closing date (it's a sanity check). This helps avoid mistakes in data entry.
- It helps the reporting code figure out which accounts are still active and filter out closed accounts outside of the reporting period. This is especially useful as your ledger accumulates much data over time, as there will be accounts that stop existing and which you just don't want to see in reports for years that follow their closure.

Note that a Close directive does not currently generate an implicit zero balance check. You may want to add one just before the closing date to ensure that the account is correctly closed with empty contents.

At the moment, once an account is closed, you cannot reopen it after that date. (Though you can, of course, delete or comment-out the directive that closed it.) Finally, there are utility functions in the code that allow you to establish which accounts are open on a particular date. I strongly recommend that you close accounts when they actually close in reality, it will keep your ledger more tidy.

Commodity

There is a "Commodity" directive that can be used to declare currencies, financial instruments, commodities (different names for the same thing in Beancount):

```
1867-07-01 commodity CAD
```

The general format of the Commodity directive is:

YYYY-MM-DD ***commodity*** *Currency*

This directive is a late arrival, and is entirely optional: you can use commodities without having to really declare them this way. The purpose of this directive is to attach commodity-specific metadata fields on it, so that it can be gathered by plugins later on. For example, you might want to provide a long descriptive name for each commodity, such as "Swiss Franc" for commodity "CHF", or "Hooli Corporation Class C Shares" for "HOOL", like this:

```
1867-07-01 commodity CAD
  name: "Canadian Dollar"
  asset-class: "cash"

2012-01-01 commodity HOOL
  name: "Hooli Corporation Class C Shares"
  asset-class: "stock"
```

For example, a plugin could then gather the metadata attribute names and perform some aggregations per asset class.

You can use any date for a commodity… but a relevant date is the date at which it was created or introduced, e.g. Canadian dollars were first introduced in 1867, ILS (new Israeli Shekels) are in use since 1986-01-01. For a company, the date the corporation was formed and shares created could be a good date. Since the main purpose of this directive is to collect per-commodity information, the particular date you choose doesn't matter all that much.

It is an error to declare the same commodity twice.

Transactions

Transactions are the most common type of directives that occur in a ledger. They are slightly different to the other ones, because they can be followed by a list of postings. Here is an example:

```
2014-05-05 txn "Cafe Mogador" "Lamb tagine with wine"
  Liabilities:CreditCard:CapitalOne         -37.45 USD
  Expenses:Restaurant
```

As for all the other directives, a transaction directive begins with a date in the *YYYY-MM-DD* format and is followed by the directive name, in this case, "`txn`". However, because transactions are the *raison d'être* for our double-entry system and as such are by far the most common type of directive that should appear in a Beancount input file, we make a special case and allow the user to elide the "txn" keyword and just use a flag instead of it:

```
2014-05-05 * "Cafe Mogador" "Lamb tagine with wine"
  Liabilities:CreditCard:CapitalOne         -37.45 USD
  Expenses:Restaurant
```

A flag is used to indicate the status of a transaction, and the particular meaning of the flag is yours to define. We recommend using the following interpretation for them:

- ***:** Completed transaction, known amounts, "this looks correct."
- **!:** Incomplete transaction, needs confirmation or revision, "this looks incorrect."

In the case of the first example using "txn" to leave the transaction unflagged, the default flag ("*") will be set on the transaction object. (I nearly always use the "*" variant and never the keyword one, it is mainly there for consistency with all the other directive formats.)

You can also attach flags to the postings themselves, if you want to flag one of the transaction's legs in particular:

```
2014-05-05 * "Transfer from Savings account"
  Assets:MyBank:Checking            -400.00 USD
  ! Assets:MyBank:Savings
```

This is useful in the intermediate stage of de-duping transactions (see Getting Started document for more details).

The general format of a Transaction directive is:

*YYYY-MM-DD [**txn**|Flag] [[Payee] Narration]*
  [Flag] Account   Amount [{Cost}] [@ Price]
  [Flag] Account   Amount [{Cost}] [@ Price]
  ...

The lines that follow the first line are for "Postings." You can attach as many postings as you want to a transaction. For example, a salary entry might look like this:

```
2014-03-19 * "Acme Corp" "Bi-monthly salary payment"
  Assets:MyBank:Checking             3062.68 USD     ; Direct deposit
  Income:AcmeCorp:Salary            -4615.38 USD     ; Gross salary
  Expenses:Taxes:TY2014:Federal       920.53 USD     ; Federal taxes
  Expenses:Taxes:TY2014:SocSec        286.15 USD     ; Social security
  Expenses:Taxes:TY2014:Medicare       66.92 USD     ; Medicare
  Expenses:Taxes:TY2014:StateNY       277.90 USD     ; New York taxes
  Expenses:Taxes:TY2014:SDI             1.20 USD     ; Disability insurance
```

The Amount in "Postings" can also be an arithmetic expression using `( ) * / - +` . For example,

```
2014-10-05 * "Costco" "Shopping for birthday"
  Liabilities:CreditCard:CapitalOne         -45.00          USD
  Assets:AccountsReceivable:John            ((40.00/3) + 5) USD
  Assets:AccountsReceivable:Michael         40.00/3         USD
  Expenses:Shopping
```

The crucial and only constraint on postings is that the sum of their balance amounts must be zero. This is explained in full detail below.

Metadata

It's also possible to attach metadata to the transaction and/or any of its postings, so the fully general format is:

```
YYYY-MM-DD [txn|Flag] [[Payee] Narration]
  [Key: Value]
  ...
  [Flag] Account      Amount [{Cost}] [@ Price]
    [Key: Value]
    ...
  [Flag] Account      Amount [{Cost}] [@ Price]
    [Key: Value]
    ...
  ...
```

See the dedicated section on metadata below.

Payee & Narration

A transaction may have an optional "payee" and/or a "narration." In the first example above, the payee is "Cafe Mogador" and the narration is "Lamb tagine with wine".

The payee is a string that represents an external entity that is involved in the transaction. Payees are sometimes useful on transactions that post amounts to Expense accounts, whereby the account accumulates a category of expenses from multiple businesses. A good example is "`Expenses:Restaurant`", which will include all postings for the various restaurants one might visit.

A narration is a description of the transaction that you write. It can be a comment about the context, the person who accompanied you, some note about the product you bought... whatever you want it to be. It's for you to insert whatever you like. I like to insert notes when I reconcile, it's quick and I can refer to my notes later on, for example, to answer the question "What was the name of that great little sushi place I visited with Andreas on the West side last winter?"

If you place a single string on a transaction line, it becomes its narration:

```
2014-05-05 * "Lamb tagine with wine"
   …
```

If you want to set just a payee, put an empty narration string:

```
2014-05-05 * "Cafe Mogador" ""
   …
```

For legacy reasons, a pipe symbol ("|") is accepted between those strings (but this will be removed at some point in the future):

```
2014-05-05 * "Cafe Mogador" | ""
   …
```

You may also leave out either (but you must provide a flag):

```
2014-05-05 *
   …
```

Note for Ledger users. Ledger does not have separate narration and payee fields, it has only one field, which is referred to by the "Payee" metadata tag, and the narration ends up in a saved comment (a "persistent note"). In Beancount, a Transaction object simply has two fields: payee and narration, where payee just happens to have an empty value a lot of the time.

For a deeper discussion of how and when to use payees or not, see Payees, Subaccounts, and Assets.

Costs and Prices

Postings represent a single amount being deposited to or withdrawn from an account. The simplest type of posting includes only its amount:

```
2012-11-03 * "Transfer to pay credit card"
  Assets:MyBank:Checking            -400.00 USD
  Liabilities:CreditCard             400.00 USD
```

If you converted the amount from another currency, you must provide a conversion rate to balance the transaction (see next section). This is done by attaching a "price" to the posting, which is the rate of conversion:

```
2012-11-03 * "Transfer to account in Canada"
  Assets:MyBank:Checking            -400.00 USD @ 1.09 CAD
  Assets:FR:SocGen:Checking          436.01 CAD
```

You could also use the "@@" syntax to specify the total cost:

```
2012-11-03 * "Transfer to account in Canada"
  Assets:MyBank:Checking            -400.00 USD @@ 436.01 CAD
  Assets:FR:SocGen:Checking          436.01 CAD
```

Beancount will automatically compute the per-unit price, that is 1.090025 CAD (note that the precision will differ between the last two examples).

After the transaction, we are not interested in keeping track of the exchange rate for the units of USD deposited into the account; those units of "USD" are simply deposited. In a sense, the rate at which they were converted at has been forgotten.

However, some commodities that you deposit in accounts must be "held at cost." This happens when you want to keep track of the *cost basis* of those commodities. The typical use case is investing, for example when you deposit shares of a stock in an account. What you want in that circumstance is for the acquisition cost of the commodities you deposit to be attached to the units, such that when you remove those units later on (when you sell), you should be able to identify by cost which of those units to remove, in order to control the effect of taxes (and avoid errors). You can imagine that for each of the units in the account there is an attached cost basis. This will allow us to automatically compute capital gains later.

In order to specify that a posting to an account is to be held at a specific cost, include the cost in curly brackets:

```
2014-02-11 * "Bought shares of S&P 500"
  Assets:ETrade:IVV                10 IVV {183.07 USD}
  Assets:ETrade:Cash         -1830.70 USD
```

This is the subject of a deeper discussion. Refer to "How Inventories Work" and "Trading with Beancount" documents for an in-depth discussion of these topics.

Finally, you can include both a cost and a price on a posting:

```
2014-07-11 * "Sold shares of S&P 500"
  Assets:ETrade:IVV               -10 IVV {183.07 USD} @ 197.90 USD
  Assets:ETrade:Cash          1979.90 USD
  Income:ETrade:CapitalGains
```

The price will only be used to insert a price entry in the prices database (see Price section below).

This is discussed in more details in the Balancing Postings section of this document.

Important Note. Amounts specified as either per-share or total prices or costs are *always unsigned*. It is an error to use a negative sign or a negative cost and Beancount will raise an error if you attempt to do so.

Balancing Rule - The "weight" of postings

A crucial aspect of the double-entry method is ensuring that the sum of all the amounts on its postings equals ZERO, in all currencies. This is the central, non-negotiable condition that engenders the accounting equation, and makes it possible to filter any subset of transactions and drawing balance sheets that balance to zero.

But with different types of units, the previously introduced price conversions and units "held at cost", what does it all mean?

It's simple: we have devised a simple and clear rule for obtaining an amount and a currency from each posting, to be used for balancing them together. We call this the "weight" of a posting, or the balancing amount. Here is a short example of weights derived from postings using the four possible types of cost/price combinations:

```
YYYY-MM-DD
  Account       10.00 USD                         -> 10.00 USD
  Account       10.00 CAD @ 1.01 USD              -> 10.10 USD
  Account       10 SOME {2.02 USD}                -> 20.20 USD
  Account       10 SOME {2.02 USD} @ 2.50 USD     -> 20.20 USD
```

Here is the explanation of how it is calculated:

1. If the posting has **only an amount** and no cost, the balance amount is just the amount and currency on that posting. Using the first example from the previous section, the amount is -400.00 USD, and that is balanced against the 400.00 USD of the second leg.
2. If the posting has **only a price**, the price is multiplied by the number of units and the price currency is used. In the second example from the preceding section, that is -400.00 USD x 1.09 CAD(/USD) = -436.00 CAD, and that is balanced against the other posting of 436.00 CAD[2].
3. If the posting **has a cost**, the cost is multiplied by the number of units and the cost currency is used. In the third example from the preceding section, that is 10 IVV x 183.08 USD(/IVV) = 1830.70 USD. That is balanced against the cash leg of -1830.70 USD, so all is good.
4. Finally, if a posting **has both a cost and a price**, we simply ignore the price. This optional price is used later on to generate an entry in the in-memory prices database, but it is not used in balancing at all.

With this rule, you should be able to easily balance all your transactions. Moreover, this rule makes it possible to let Beancount automatically calculate capital gains for you (see Trading with Beancount for details).

Reducing Positions

When you post a *reduction* to a position in an account, the reduction must always match an existing lot. For example, if an account holds 3200 USD and a transaction posts a -1200 USD change to that

[2] Note that this is valid whether the price is specified as a per-unit price with the @ syntax or as a total price using the @@ syntax.

account, the 1200 USD match against the existing 3200 USD, and the result is a single position of 2000 USD. This also works for negative values. For example, if an account has a -1300 USD balance and you post a +2000 USD change to it, you obtain a 700 USD balance.

A change posted to an account, regardless of the account type, can result in a positive or negative balance; there are no limitations on the balances of simple commodity amounts (that is, those with no cost associated to them). For example, while Assets accounts *normally* have a positive balance and Liabilities accounts *usually* a negative one, you can legally credit an Assets account to a negative balance, or debit a Liabilities account to a positive balance. This is because in the real world these things do happen: you might write a check too many and obtain temporary credit from your bank's checking account (usually along with an outrageous "overdraft" fee), or pay that credit card balance twice by mistake.

For commodities held at cost, the cost specification of the posting must match one of the lots held in the inventory before the transaction is applied. The list of lots is gathered, and matched against the specification in the {...} part of the posting. For example, if you provide a cost, only those lots whose cost match that will remain. If you provide a date, only those lots which match that date will remain. And you can use a label as well. If you provide a cost and a date, both of these are matched against the list of candidate lots to reduce. This is essentially a filter on the list of lots.

If the filtered list results in a single lot, that lot is chosen to be reduced. If the list results in multiple lots, but the total amount being reduced equals the total amount in the lots, all those lots are reduced by that posting.

For example, if in the past you had the following transactions:

```
2014-02-11 * "Bought shares of S&P 500"
  Assets:ETrade:IVV                20 IVV {183.07 USD, "ref-001"}
  …

2014-03-22 * "Bought shares of S&P 500"
  Assets:ETrade:IVV                15 IVV {187.12 USD}
  …
```

Each of the following reductions would be unambiguous:

```
2014-05-01 * "Sold shares of S&P 500"
  Assets:ETrade:IVV               -20 IVV {183.07 USD}
  …

2014-05-01 * "Sold shares of S&P 500"
  Assets:ETrade:IVV               -20 IVV {2014-02-11}
  …

2014-05-01 * "Sold shares of S&P 500"
  Assets:ETrade:IVV               -20 IVV {"ref-001"}
  …

2014-05-01 * "Sold shares of S&P 500"
  Assets:ETrade:IVV               -35 IVV {}
  …
```

However, the following would be ambiguous:

```
2014-05-01 * "Sold shares of S&P 500"
  Assets:ETrade:IVV               -20 IVV {}
  …
```

If multiple lots match against the reducing posting and their number is not the total number, we are in a situation of *ambiguous matches*. What happens then, is that the account's *booking method* is invoked. There are multiple booking methods, but by default, all accounts are set to use the "STRICT" booking method. This method simply issues an error in an ambiguous situation.

You may set the account's booking method to "FIFO" to instruct Beancount to select the oldest of the lots. Or "LIFO" for the latest (youngest) of the lots. This will automatically select all the necessary matching lots to fulfill the reduction.

PLEASE NOTE! Requiring the dates to match will be dealt with more sensibly in the near future. See A Proposal for an Improvement on Inventory Booking for details of this upcoming change.

For such postings, a change that results in a negative number of units is usually impossible. Beancount does not currently allow holding a negative number of a commodity held at cost. For example, an input with just this transaction will fail:

```
2014-05-23 *
  Assets:Investments:MSFT        -10 MSFT {43.40 USD}
  Assets:Investments:Cash     434.00 USD
```

If it did not, this would result in a balance of -10 units of MSFT. On the other hand, if the account had a balance of 12 units of MSFT held at 43.40 USD on 5/23, the transaction would book just fine, reducing the existing 12 units to 2. Most often, the error that will occur is that the account will be holding a balance of 10 or more units of MSFT at a *different cost*, and the user will specify an incorrect value for the cost. For instance, if the account had a positive balance of 20 MSFT {42.10 USD}, the transaction above would still fail, because there aren't 10 or more units of MSFT at 43.40 USD to remove from.

This constraint is enforced for a few reasons:

- Mistakes in data entry for stock units are not uncommon, they have an important negative impact on the correctness of your Ledger—the amounts are usually large—and they will typically trigger an error from this constraint. Therefore, the error check is a useful way to detect these types of errors.
- Negative numbers of units held at cost are fairly rare. Chances are you don't need them at all. Exceptions include: short sales of stock, holding spreads on futures contracts, and depending on how you account for them, short positions in currency trading.

This is why this check is enabled by default.

PLEASE NOTE! In a future version of Beancount, we will relax this constraint somewhat. We will allow an account to hold a negative number of units of a commodity if and only if there are no other units of that commodity held in the account. Either that, or we will allow you to mark an account as having no such constraints at all. The purpose is to allow the account of short positions in commodities. The only blocking factor is this constraint.

For more details of the inventory booking algorithm, see the How Inventories Work document.

Amount Interpolation

Beancount is able to fill in some of the details of a transaction automatically. You can currently elide the amount of *at most* one posting within a transaction:

```
2012-11-03 * "Transfer to pay credit card"
  Assets:MyBank:Checking            -400.00 USD
  Liabilities:CreditCard
```

In the example above, the amount of the credit card posting has been elided. It is automatically calculated by Beancount at 400.00 USD to balance the transaction. This also works with multiple postings, and with postings with costs:

```
2014-07-11 * "Sold shares of S&P 500"
  Assets:ETrade:IVV                 -10 IVV {183.07 USD}
  Assets:ETrade:Cash            1979.90 USD
  Income:ETrade:CapitalGains
```

In this case, the units of IVV are sold at a higher price ($197.90) than they were bought for ($183.07). The cash first posting has a weight of -10 x 183.07 = -1830.70 and the second posting a straightforward $1979.90. The last posting will be ascribed the difference, that is, a balance of `-149.20 USD`, which is to say, a *gain* of $149.20.

When calculating the amount to be balanced, the same balance amounts that are used to check that the transaction balances to zero are used to fill in the missing amounts. For example, the following would not trigger an error:

```
2014-07-11 * "Sold shares of S&P 500"
  Assets:ETrade:IVV                 -10 IVV {183.07 USD} @ 197.90 USD
  Assets:ETrade:Cash
```

The cash account would receive 1830.70 USD automatically, because the balance amount of the IVV posting is -1830.70 USD (if a posting has both a cost and a price, the cost basis is always used and the optional price ignored). While this is accepted and correct from a balancing perspective, this would be incomplete from an accounting perspective: the capital gain on a sale needs to be accounted for separately and besides, the amount deposited to the cash account if you did as above would fall short of the real deposit (1979.00 USD) and hopefully a subsequent balance assertion in the cash account would indicate this oversight by triggering an error.

Finally, this also works when the balance includes multiple commodities:

```
2014-07-12 * "Uncle Bob gave me his foreign currency collection!"
  Income:Gifts                 -117.00 ILS
  Income:Gifts                -3000.00 INR
  Income:Gifts                 -800.00 JPY
  Assets:ForeignCash
```

Multiple postings (one for each commodity required to balance) will be inserted to replace the elided one.

Tags

Transactions can be tagged with arbitrary strings:

```
2014-04-23 * "Flight to Berlin" #berlin-trip-2014
  Expenses:Flights              -1230.27 USD
```

```
    Liabilities:CreditCard
```

This is similar to the popular idea of “hash tagging” on Twitter and such. These tags essentially allow you to mark a subset of transactions. They can then be used as a filter to generate reports on only this subset of transactions. They have numerous uses. I like to use them to mark all my trips.

Multiple tags can be specified as well:

```
2014-04-23 * "Flight to Berlin" #berlin-trip-2014 #germany
  Expenses:Flights              -1230.27 USD
  Liabilities:CreditCard
```

(If you want to store key-value pairs on directives, see the section on metadata below.)

The Tag Stack

Oftentimes multiple transactions related to a single tag will be entered consecutively in a file. As a convenience, the parser can automatically tag transactions within a block of text. How this works is simple: the parser has a “stack” of current tags which it applies to all transactions as it reads them one-by-one. You can push and pop tags onto/from this stack, like this:

```
pushtag #berlin-trip-2014

2014-04-23 * "Flight to Berlin"
  Expenses:Flights              -1230.27 USD
  Liabilities:CreditCard

poptag #berlin-trip-2014
```

This way, you can also push multiple tags onto a long, consecutive set of transactions without having to type them all in.

Links

Transactions can also be linked together. You may think of the link as a special kind of tag that can be used to group together a set of financially related transactions over time. For example you may use links to group together transactions that are each related with a specific invoice. This allows to track payments (or write-offs) associated with the invoice:

```
2014-02-05 * "Invoice for January" ^invoice-pepe-studios-jan14
  Income:Clients:PepeStudios           -8450.00 USD
  Assets:AccountsReceivable

2014-02-20 * "Check deposit - payment from Pepe" ^invoice-pepe-studios-jan14
  Assets:BofA:Checking                  8450.00 USD
  Assets:AccountsReceivable
```

Or track multiple transfers related to a single nefarious purpose:

```
2014-02-05 * "Moving money to Isle of Man" ^transfers-offshore-17
  Assets:WellsFargo:Savings          -40000.00 USD
  Assets:WellsFargo:Checking          40000.00 USD

2014-02-09 * "Wire to FX broker" ^transfers-offshore-17
  Assets:WellsFargo:Checking         -40025.00 USD
  Expenses:Fees:WireTransfers            25.00 USD
  Assets:OANDA:USDollar               40000.00
```

```
2014-03-16 * "Conversion to offshore beans" ^transfers-offshore-17
  Assets:OANDA:USDollar          -40000.00 USD
  Assets:OANDA:GBPounds           23391.81 GBP @ 1.71 USD

2014-03-16 * "God save the Queen (and taxes)" ^transfers-offshore-17
  Assets:OANDA:GBPounds             -23391.81 GBP
  Expenses:Fees:WireTransfers           15.00 GBP
  Assets:Brittania:PrivateBanking    23376.81 GBP
```

Linked transactions can be rendered by the web interface in their own dedicated journal, regardless of the current view/filtered set of transactions (the list of links is a global page).

Balance Assertions

A balance assertion is a way for you to input your statement balance into the flow of transactions. It tells Beancount to verify that the number of units of a particular commodity in some account should equal some expected value at some point in time. For instance, this

```
2014-12-26 balance Liabilities:US:CreditCard   -3492.02 USD
```

says "Check that the number of USD units in account "`Liabilities:US:CreditCard`" on the *morning* of December 26th, 2014 is -3492.02 USD." When processing the list of entries, if Beancount encounters a different balance than this for USD it will report an error.

If no error is reported, you should have some confidence that the list of transactions that precedes it in this account is highly likely to be correct. This is useful in practice, because in many cases some transactions can get imported separately from the accounts of each of their postings (see the de-duping problem). This can result in you booking the same transaction twice without noticing, and regularly inserting a balance assertion will catch that problem every time.

The general format of the Balance directive is:

*YYYY-MM-DD **balance** Account Amount*

Note that a balance assertion, like all other non-transaction directives, applies at the **beginning** of its date (i.e., midnight at the start of day). Just imagine that the balance check occurs right after midnight on that day.

Balance assertions only make sense on balance sheet accounts (Assets and Liabilities). Because the postings on Income and Expenses accounts are only interesting because of their transient value, i.e., for these accounts we're interested in sums of changes over a period of time (not the absolute value), it makes little sense to use a balance assertion on income statement accounts.

Also, take note that each account benefits from an implicit balance assertion that the account is empty after it is opened at the date of its Open directive. You do not need to explicitly assert a zero balance when opening accounts.

Multiple Commodities

A Beancount account may contain more than one commodity (although in practice, you will find that this does not occur often, it is sensible to create dedicated sub-accounts to hold each commodity, for example, holding a portfolio of stocks). A balance assertion applies *only* to the commodity of the assertion; it leaves the other commodities in the balance unchecked. If you want to check multiple commodities, use multiple balance assertions, like this:

```
; Check cash balances from wallet
```

```
2014-08-09 balance Assets:Cash     562.00 USD
2014-08-09 balance Assets:Cash     210.00 CAD
2014-08-09 balance Assets:Cash      60.00 EUR
```

There is currently no way to exhaustively check the full list of commodities in an account (a proposal is underway).

Note that in this example if an exhaustive check really matters to you, you could circumvent by defining a subaccount of the cash account to segregate each commodity separately, like this `Assets:Cash:USD`, `Assets:Cash:CAD`.

Lots Are Aggregated

The balance assertion applies to the sum of units of a particular commody, irrespective of their cost. For example, if you hold three lots of the same commodity in an account, for example, 5 HOOL {500 USD} and 6 HOOL {510 USD}, the following balance check should succeed:

```
2014-08-09 balance Assets:Investing:HOOL     11 HOOL
```

All the lots are aggregated together and you can verify their number of units.

Checks on Parent Accounts

Balance assertions may be performed on parent accounts, and will include the balances of theirs and their sub-accounts:

```
2014-01-01 open Assets:Investing
2014-01-01 open Assets:Investing:Apple       AAPL
2014-01-01 open Assets:Investing:Amazon      AMZN
2014-01-01 open Assets:Investing:Microsoft   MSFT
2014-01-01 open Equity:Opening-Balances

2014-06-01 *
  Assets:Investing:Apple       5 AAPL {578.23 USD}
  Assets:Investing:Amazon      5 AMZN {346.20 USD}
  Assets:Investing:Microsoft   5 MSFT {42.09 USD}
  Equity:Opening-Balances

2014-07-13 balance Assets:Investing 5 AAPL
2014-07-13 balance Assets:Investing 5 AMZN
2014-07-13 balance Assets:Investing 5 MSFT
```

Note that this does require that a parent account have been declared as Open, in order to be legitimately used in the balance assertions directive.

Before Close

It is useful to insert a balance assertion for 0 units just before closing an account, just to make sure its contents are empty as you close it. The Close directive does not insert that for you automatically (we may eventually build a plug-in for it).

Local Tolerance

It's pretty common that sometimes one needs to override the tolerance on the balance check to loosen it on that balance assertion. This can be done using a local tolerance amount off of the balance amount, like this:

```
2013-09-20 balance Assets:Investing:Funds     319.020 ~ 0.002 RGAGX
```

Pad

A padding directive automatically inserts a transaction that will make the subsequent balance assertion succeed, if it is needed. It inserts the difference needed to fulfill that balance assertion. (What "rubber space" is in LaTeX, Pad directives are to balances in Beancount.)

Note that by "subsequent," I mean in *date order*, not in the order of the declarations in the file. This is the conceptual equivalent of a transaction that will automatically expand or contract to fill the difference between two balance assertions over time. It looks like this:

```
2014-06-01 pad Assets:BofA:Checking Equity:Opening-Balances
```

The general format of the Pad directive is:

*YYYY-MM-DD **pad** Account AccountPad*

The first account is the account to credit the automatically calculated amount to. This is the account that should have a balance assertion following it (if the account does not have a balance assertion, the pad entry is benign and does nothing).

The second account is for the other leg of the transaction, it is the source where the funds will come from, and this is almost always some Equity account. The reason for this is that this directive is generally used for initializing the balances of new accounts, to save us from having to either insert such a directive manually, or from having to enter the full past history of transactions that will bring the account to its current balance.

Here is a realistic example usage scenario:

```
; Account was opened way back in the past.
2002-01-17 open Assets:US:BofA:Checking

2002-01-17 pad Assets:US:BofA:Checking Equity:Opening-Balances

2014-07-09 balance Assets:US:BofA:Checking  987.34 USD
```

This will result in the following transaction being inserted right after the Pad directive, on the same date:

```
2002-01-17 P "(Padding inserted for balance of 987.34 USD)"
  Assets:US:BofA:Checking        987.34 USD
  Equity:Opening-Balances       -987.34 USD
```

This is a normal transaction—you will see it appear in the rendered journals along with the other ones. (Note the special "P" flag, which can be used by scripts to find these.)

Observe that without balance assertions, Pad directives make no sense. Therefore, like balance assertions, they are normally only used on balance sheet accounts (Assets and Liabilities).

You could also insert Pad entries *between* balance assertions, it works too. For example:

```
2014-07-09 balance Assets:US:BofA:Checking  987.34 USD
… more transactions…
2014-08-08 pad Assets:US:BofA:Checking Equity:Opening-Balances

2014-08-09 balance Assets:US:BofA:Checking  1137.23 USD
```

Without intervening transactions, this would insert the following padding transaction:

```
2014-08-08 P "(Padding inserted for balance of 1137.23 USD)"
  Assets:US:BofA:Checking        149.89 USD
  Equity:Opening-Balances       -149.89 USD
```

In case that's not obvious, 149.89 USD is the difference between 1137.23 USD and 987.34 USD. If there were more intervening transactions posting amounts to the checking account, the amount would automatically have been adjusted to make the second assertion pass.

Unused Pad Directives

You may not currently leave unused Pad directives in your input file. They will trigger an error:

```
2014-01-01 open Assets:US:BofA:Checking

2014-02-01 pad Assets:US:BofA:Checking Equity:Opening-Balances

2014-06-01 * "Initializing account"
  Assets:US:BofA:Checking                    212.00 USD
  Equity:Opening-Balances

2014-07-09 balance Assets:US:BofA:Checking  212.00 USD
```

(Being this strict is a matter for debate, I suppose, and it could eventually be moved to an optional plugin.)

Commodities

Note that the Pad directive does not specify any commodities at all. All commodities with corresponding balance assertions in the account are affected. For instance, the following code would have a padding directive insert a transaction with separate postings for USD and CAD:

```
2002-01-17 open Assets:Cash

2002-01-17 pad Assets:Cash Equity:Opening-Balances

2014-07-09 balance Assets:Cash    987.34 USD
2014-07-09 balance Assets:Cash    236.24 CAD
```

If the account contained other commodities that aren't balance asserted, no posting would be inserted for those.

Cost Basis

At the moment, Pad directives do not work with accounts holding positions held at cost. The directive is really only useful for cash accounts. (This is mainly because balance assertions do not yet allow specifying a cost basis to assert. It is possible that in the future we decide to support asserting the total cost basis, and that point we could consider supporting padding with cost basis.)

Multiple Paddings

You cannot currently insert multiple padding entries for the same account and commodity:

```
2002-01-17 open Assets:Cash

2002-02-01 pad Assets:Cash Equity:Opening-Balances

2002-03-01 pad Assets:Cash Equity:Opening-Balances
```

```
2014-04-19 balance Assets:Cash    987.34 USD
```

(There is a proposal pending to allow this and spread the padding amount evenly among all the intervening Pad directives, but it is as of yet unimplemented.)

Notes

A Note directive is simply used to attach a dated comment to the journal of a particular account, like this:

```
2013-11-03 note Liabilities:CreditCard "Called about fraudulent card."
```

When you render the journal, the note should be rendered in context. This can be useful to record facts and claims associated with a financial event. I often use this to record snippets of information that would otherwise not make their way to a transaction.

The general format of the Note directive is:

YYYY-MM-DD ***note*** *Account Description*

The description string may be split among multiple lines.

Documents

A Document directive can be used to attach an external file to the journal of an account:

```
2013-11-03 document Liabilities:CreditCard "/home/joe/stmts/apr-2014.pdf"
```

The filename gets rendered as a browser link in the journals of the web interface for the corresponding account and you should be able to click on it to view the contents of the file itself. This is useful to integrate account statements and other downloads into the flow of accounts, so that they're easily accessible from a few clicks. Scripts could also be written to obtain this list of documents from an account name and do something with them.

The general format of the Document directive is:

YYYY-MM-DD ***document*** *Account PathToDocument*

Documents from a Directory

A more convenient way to create these entries is to use a special option to specify directories that contain a hierarchy of sub-directories that mirrors that of the chart of accounts. For example, the Document directive shown in the previous section could have been created from a directory hierarchy that looks like this:

```
stmts
`-- Liabilities
    `-- CreditCard
        `-- 2014-04-27.apr-2014.pdf
```

By simply specifying the root directory as an option (note the absence of a trailing slash):

```
option "documents" "/home/joe/stmts"
```

The files that will be picked up are those that begin with a date as above, in the *YYYY-MM-DD* format. You may specify this option multiple times if you have many such document archives. In the past I have used one directory for each year (if you scan all your documents, the directory can grow to a large size, scanned documents tend to be large files).

Organizing your electronic document statements and scans using the hierarchy of your ledger's accounts is a fantastic way to organize them and establish a clear, unambiguous place to find these documents later on, when they're needed.

Prices

Beancount sometimes creates an in-memory data store of prices for each commodity, that is used for various reasons. In particular, it is used to report unrealized gains on account holdings. Price directives can be used to provide data points for this database. A Price directive establishes the rate of exchange between one commodity (the base currency) and another (the quote currency):

```
2014-07-09 price HOOL  579.18 USD
```

This directive says: "The price of one unit of HOOL on July 9th, 2014 was 579.18 USD." Price entries for currency exchange rates work the same way:

```
2014-07-09 price USD  1.08 CAD
```

The general format of the Price directive is:

*YYYY-MM-DD **price** Commodity Price*

Remember that Beancount knows nothing about what HOOL, USD or CAD are. They are opaque "things." You attach meaning to them. So setting a price per hour for your vacation hours is perfectly valid and useful too, if you account for your unused vacations on such terms:

```
2014-07-09 price VACHR  38.46 USD  ; Equiv. $80,000 year
```

Prices from Postings

If you use the `beancount.plugins.implicit_prices` plugin, every time a Posting appears that has a cost or an optional price declared, it will use that cost or price to automatically synthesize a Price directive. For example, this transaction:

```
2014-05-23 *
  Assets:Investments:MSFT        -10 MSFT {43.40 USD}
  Assets:Investments:Cash     434.00 USD
```

automatically becomes this after parsing:

```
2014-05-23 *
  Assets:Investments:MSFT        -10 MSFT {43.40 USD}
  Assets:Investments:Cash     434.00 USD

2014-05-23 price MSFT  43.40 USD
```

This is convenient and if you enable it, will probably be where most of the price points in your ledger's price database will come from. You can print a table of the parsed prices from a ledger (it is just another type of report).

Prices on the Same Day

Notice that there is no notion of time; Beancount is not designed to solve problems for intra-day traders, though of course, it certainly is able to handle multiple trades per day. It just stores its prices per day. (Generally, if you need many features that require a notion of intra-day time, you're better off using another system, this is not the scope of a bookkeeping system.)

When multiple Price directives do exist for the same day, the last one to appear in the file will be selected for inclusion in the Price database.

Events

Event directives[3] are used to track the value of *some variable of your choice* over time. For example, your location:

```
    2014-07-09 event "location" "Paris, France"
```

The above directive says: "Change the value of the 'location' event to 'Paris, France' as of the 9th of July 2014 and onwards." A particular event type only has a single value per day.

The general format of the Event directive is:

YYYY-MM-DD ***event*** *Name Value*

The event's *Name* string does not have to be declared anywhere, it just begins to exist the first time you use the directive. The *Value* string can be anything you like; it has no prescribed structure.

How to use these is best explained by providing examples:

- **Location**: You can use events for tracking the city or country you're in. This is sensible usage within a ledger because the nature of your expenses are heavily correlated to where you are. It's convenient: if you use Beancount regularly, it is very little effort to add this bit of information to your file. I usually have a "cash daybook" section where I put miscellaneous cash expenses, and that's where I enter those directives.
- **Address**: If you move around a lot, it's useful to keep a record of your past home addresses. This is sometimes requested on government forms for immigration. The Green Card application process in the US, for instance,
- **Employer**: You can record your date of employment and departure for each job this way. Then you can count the number of days you worked there.
- **Trading window**: If you're an employee of a public company, you can record the dates that you're allowed to trade its stock. This can then be used to ensure you did not trade that stock when the window is closed.

Events are often used to report on numbers of days. For example, in the province of Quebec (Canada) you are insured for free health care coverage if "you spend 183 days of the calendar year or more, excluding trips of 21 days or less." If you travel abroad a lot, you could easily write a script to warn you of remaining day allocations to avoid losing your coverage. Many expats are in similar situations.

In the same vein, the IRS recognizes US immigrants as "resident alien"—and thus subject to filing taxes, yay!—if you pass the Substantial Presence Test, which is defined as "being physically present in the US on at least 31 days during the current year, and 193 days during the 3 year period that includes the current year and the 2 years immediately before that, counting: all the days you were present in the current year, and ⅓ of the days of the year before that, and ⅙ of the year preceding that one." Ouch… my head hurts. This gets complicated. Armed with your location over time, you could report it automatically.

Events will also be usable in the filtering language, to specify non-contiguous periods of time. For example, if you are tracking your location using Event directives, you could produce reports for transactions that occur only when you are in a specific location, e.g., "Show me my expenses on all

[3] I really dislike the name "event" for this directive. I've been trying to find a better alternative, so far without success. The name "register" might be more appropriate, as it resembles that of a processor's register, but that could be confused with an *account register* report. "variable" might work, but somehow that just sounds too computer-sciency and out of context. If you can think of something better, please make a suggestion and I'll seriously entertain a complete rename (with legacy support for "event").

my trips to Germany," or "Give me a list of payees for restaurant transactions when I'm in Montreal."

PLEASE NOTE! Filters haven't been implemented yet. Also, reports on events have not yet been re-implemented in Beancount 2.0. They will be reintroduced again soon, as well as filtering.

It is worth noticing that the Price and Event directives are the only ones not associated to an account.

Query

It can be convenient to be able to associate SQL queries in a Beancount file to be able to run these as a report automatically. This is still an early development / experimental directive. In any case, you can insert queries in the stream of transactions like this:

```
2014-07-09 query "france-balances" "
  SELECT account, sum(position) WHERE ‘trip-france-2014’ in tags"
```

The grammar is

*YYYY-MM-DD **query** Name SqlContents*

Each query has a name, which will probably be used to invoke its running as a report type. Also, the date of the query is intended to be the date at which the query is intended to be run for, that is, transactions following it should be ignored. If you're familiar with the SQL syntax, it's an implicit CLOSE.

Custom

The long-term plan for Beancount is to allow plugins and external clients to define their own directive types, to be declared and validated by the Beancount input language parser. In the meantime, a generic directive is provided for clients to **prototype** new features, e.g., budgeting.

```
2014-07-09 custom "budget" "..." TRUE 45.30 USD
```

The grammar for this directive is flexible:

*YYYY-MM-DD **custom** TypeName Value1 ...*

The first argument is a string and is intended to be unique to your directive. Think of this as the type of your directive. Following it, you can put an arbitrary list of strings, dates, booleans, amounts, and numbers. Note that there is no validation that checks that the number and types of arguments following the *TypeName* is consistent for a particular type.

(See this thread for the origin story around this feature.)

Metadata

You may attach arbitrary data to each of your entries and postings. The syntax for it looks like this:

```
2013-03-14 open Assets:BTrade:HOOLI
  category: "taxable"
```

```
2013-08-26 * "Buying some shares of Hooli"
  statement: "confirmation-826453.pdf"
  Assets:BTrade:HOOLI      10 HOOL @ {498.45 USD}
    decision: "scheduled"
  Assets:BTrade:Cash
```

In this example, a "category" attribute is attached to an account's Open directive, a "statement" attribute is attached to a Transaction directive (with a string value that represents a filename) and a "decision" attribute has been attached to the first posting of the transaction (the additional indentation from the posting is not strictly necessary but it helps with readability).

Metadata can be attached to any directive type. Keys must begin with a lowercase character from a-z and may contain (uppercase or lowercase) letters, numbers, dashes and underscores. Moreover, the values can be any of the following data types:

- Strings
- Accounts
- Currency
- Dates (datetime.date)
- Tags
- Numbers (Decimal)
- Amount (beancount.core.amount.Amount)

There are two ways in which this data can be used:

1. Query tools that come with Beancount (such as bean-query) will allow you to make use of the metadata values for filtering and aggregation.
2. You can access and use the metadata in custom scripts. The metadata values are accessible as a "`.meta`" attribute on all directives and is a Python dict.

There are no special meanings attached to particular attributes, these are intended for users to define. However, all directives are guaranteed to contain a '`filename`' (string) and a '`lineno`' (integer) attribute which reflect the location they were created from.

Finally, attributes without a value will be parsed and have a value of 'None'. If an attribute is repeated multiple times, only the first value for this attribute will be parsed and retained and the following values ignored.

Options

The great majority of a Beancount input file consists in directives, as seen in the previous section. However, there are a few global options that can be set in the input file as well, by using a special undated "option" directive:

option "title" "Ed's Personal Ledger"

The general format of the Option directive is:

option *Name Value*

where *Name* and *Value* are both strings.

Note that depending the option, the effect may be to set a single value, or add to a list of existing values. In other words, some options are lists.

There are three ways to view the list of options:

- In this document, which I update regularly.
- To view the definitive list of options supported by your installed version, use the following command: `bean-doctor list-options`
- Finally, you can peek at the source code as well.

Operating Currencies

One notable option is “`operating_currency`”. By default Beancount does not treat any of the commodities any different from each other. In particular, it doesn’t know that there’s anything special about the most common of commodities one uses: their currencies. For example, if you live in New Zealand, you’re going to have an overwhelming number of NZD commodities in your transactions.

But useful reports try to reduce all non-currency commodities into one of the main currencies used. Also, it’s useful to break out the currency units into their own dedicated columns. This may also be useful for exporting in order to avoid having to specify the units for that column and import to a spreadsheet with numbers you can process.

For this reason, you are able to declare the most common currencies you use in an option:

```
option "operating_currency" "USD"
```

You may declare more than one.

In any case, this option is only ever used by reporting code, it never changes the behavior of Beancount’s processing or semantics.

Plugins

In order to load plugin Python modules, use the dedicated “plugin” directive:

```
plugin "beancount.plugins.module_name"
```

The name of a plugin should be the name of a Python module in your PYTHONPATH. Those modules will be imported by the Beancount loader and run on the list of parsed entries in order for the plugins to transform the entries or output errors. This allows you to integrate some of your code within Beancount, making arbitrary transformations on the entries. See Scripting & Plugins for details.

Plugins also optionally accept some configuration parameters. These can be provided by an optional final string argument, like this:

```
plugin "beancount.plugins.module_name" "configuration data"
```

The general format of the Option directive is:

plugin *ModuleName StringConfig*

The format of the configuration data is plugin-dependent. At the moment, an arbitrary string is passed provided to the plugin. See the plugins’ documentation for specific detail on what it can accept.

Also see the “plugin processing mode” option which affects the list of built-in plugins that get run.

Includes

Include directives are supported. This allows you to split up large input files into multiple files. The syntax looks like this:

```
include "path/to/include/file.beancount"
```

The general format is

***include** Filename*

The specified path can be an absolute or a relative filename. If the filename is relative, it is relative to the including filename's directory. This makes it easy to put relative includes in a hierarchy of directories that can be placed under source control and checked out anywhere.

Include directives are not processed strictly (as in C, for example). The include directives are accumulated by the Beancount parser and processed separately by the loader. This is possible because the order of declarations of a Beancount input file is not relevant.

However, for the moment, options are parsed per-file. The options-map that is kept for post-parse processing is the options-map returned for the top-level file. This is probably subject to review in the future.

What's Next?

This document described all the possible syntax of the Beancount language. If you haven't written any Beancount input yet, you can head to the Getting Started guide, or browse through a list of practical use cases in the Command-line Accounting Cookbook.