A Comparison of Beancount and Ledger

[Martin Blais](http://plus.google.com/+MartinBlais), September 2014

<http://furius.ca/beancount/doc/comparison>

*The question of how Beancount differs from Ledger & HLedger has come up a few times on mailing-lists and in private emails. This document highlights key differences between these systems, as they differ sharply in their design and implementations.*

*Keep in mind that this document is written from the perspective of Beancount and as its author, reflects my own biased views for what the design of a CLI accounting system should be. My purpose here is not to shoot down other systems, but rather to highlight material differences to help newcomers understand how these systems vary in their operation and capabilities, and perhaps to stimulate a fruitful discussion about design choices with the other developers.*

[Differences](#_srg4k63oqegy)

[Inventory Booking](#_kqx4yya021of)

[Currency Conversions](#_pc2evutbfavh)

[Isolation of Inputs](#_2kt2haz5q3hv)

[Language Syntax](#_qj98uwv97n3m)

[Order Independence](#_8ifnv67670w7)

[Account Types](#_gooyvgqi7ncv)

[Transactions Must Balance](#_j1ip37rev67r)

[Numbers and Precision of Operations](#_qfxb9inw3w17)

[Filtering at the Transactional Level](#_mkqouvbjjm11)

[Extension Mechanisms](#_dbkg53hfpeyp)

[Automated Transactions via Plugins](#_7xphy1tqczro)

[No Support for Time or Effective Dates](#_egvlmsz4b5yx)

[Documents](#_ejsj91b9jx3a)

[Simpler and More Strict](#_bnwf00r21q74)

[Web Interface](#_gwqealhicxf7)

[Missing Features](#_kb3x4ggj7dnt)

[Console Output](#_k4ko9kdkm2t6)

[Filtering Language](#_9rdkeqdgzauy)

[No Meta-data](#_15c5b7d0imyo)

[No Arithmetic Expressions](#_1njrs0j9rfw0)

[Limited Support for Unicode](#_vcj70idgw3k5)

[No Forecasting or Periodic Transactions](#_j8kar5lm9buv)

# Philosophical Differences

(See [this thread](https://groups.google.com/d/msg/ledger-cli/B9qDXoSIJxQ/LVdzPkT6DAAJ).)

First, Ledger is optimistic. It assumes it's easy to input correct data by a user. My experience with data entry of the kind we're doing is that it's impossible to do this right without many automated checks. Sign errors on unasserted accounts are very common, for instance. In contrast, Beancount is highly pessimistic. It assumes the user is unreliable. It imposes a number of constraints on the input. For instance, if you added a share of AAPL at $100 to an empty account it won't let you remove a share of AAPL at $101 from it; you just don't have one. It doesn't assume the user is able or should be relied upon to input transactions in the correct order (dated assertions instead of file-order assertions). It optionally checks that proceeds match sale price (sellgains plugin). And it allows you to place extra constraints on your chart of accounts, e.g. a plugin that refuses postings to non-leaf accounts, or that refuses more than one commodity per account, or that requires you declare all accounts with Open directives; choose your level of pedanticity a-la-carte. It adds more automated cross-checks than the double-entry method provides. After all, cross-checking is why we choose to use the DE method in the first place, why not go hardcore on checking for correctness? Beancount should appeal to anyone who does not trust themselves too much. And because of this, it does not provide support for unbalanced/virtual postings; it's not a shortcoming, it's on purpose.

Secondly, there's a design ethos difference. As is evidenced in the user manual, Ledger provides a myriad of options. This surely will be appealing to many, but to me it seems it has grown into a very complicated monolithic tool. How these options interact and some of the semantic consequences of many of these options are confusing and very subtle. Beancount offers a minimalistic approach: while there are some [small number of options](https://bitbucket.org/blais/beancount/src/tip/beancount/parser/options.py?fileviewer=file-view-default#options.py-196), it tries really hard not to have them. And those options that do affect the semantics of transactions always appear in the input file (nothing on the command-line) and are distinct from the options of particular tools. Loading a file always results in the same stream of transactions, regardless of the reporting tool that will consume them. The only command-line options present are those which affect the particular behavior of the reporting tool invoked; those never change the semantics of the stream itself.

Thirdly, Beancount embraces stream processing to a deeper extent. Its loader creates a single ordered list of directives, and all the directives share some common attributes (a name, a date, metadata). This is all the data. Directives that are considered "grammar" in Ledger are defined as ordinary directive objects, e.g. "Open" is nothing special in Beancount and does nothing by itself. It's simply used in some routines that apply constraints (an account appears, has an Open directive been witnessed prior?) or that might want to hang per-account metadata to them. Prices are also specified as directives and are embedded in the stream, and can be generated in this way. All internal operations are defined as processing and outputting a stream of directives. This makes it possible to allow a user to insert their own code inside the processing pipeline to carry out arbitrary transformations on this stream of directives—anything is possible, unlimited by the particular semantics of an expression language. It's a mechanism that allows users to build new features by writing a short add-on in Python, which gets run at the core of Beancount, not an API to access its data at the edges. If anything, Beancount's own internal processing will evolve towards doing less and less and moving all the work to these plugins, perhaps even to the extent of allowing plugins to declare the directive types (with the exception of Transaction objects). It is evolving into a shallow driver that just puts together a processing pipeline to produce a stream of directives, with a handy library and functional operations.

# Specific Differences

## Inventory Booking & Cost Basis Treatment

Beancount applies strict rules regarding reductions in positions from inventories tracking the contents of its accounts. This means that you can only take out of an account something that you placed in it previously (in time), or an error will be generated. This is enforced for units “held at cost” (e.g., stock shares), to ensure that

* no cost basis can ever leak from accounts,
* we can detect errors in data entry for trades (which are all too common), and
* we are able to correctly calculate capital gains.

In contrast, Ledger does not implement inventory booking checks over time: all lots are simply accumulated regardless of the previous contents of an inventory (there is no distinction between lot addition vs. reduction). In Beancount, reductions to the contents of an inventory are required to match particular lots with a specified cost basis.

In Ledger, the output is slightly misleading about this: in order to simplify the reporting output the user may specify one of a few types of lot merging algorithms. By default, the sum of units of all lots is printed, but by using these options you can tell the reporting generation to consider the cost basis (what it calls “prices”) and/or the dates the lots were created at in which case it will report the set of lots with distinct cost bases and/or dates individually. You can select which type of merging occurs via command-line options, e.g. --lot-dates. Most importantly, this means that it is legal in Ledger to remove from an account a lot that was never added to it. This results in a mix of long and short lots, which do not accurately represent the actual changes that occur in accounts. However, it trivially allows for average cost basis reporting.

I believe that this is not only confusing, but also an [incorrect](https://groups.google.com/d/msg/ledger-cli/zRSMle5AV3Q/gAK4EbwbgXsJ) treatment of account inventories and have argued it can lead to leakage and incorrect calculations. More details and an example [are available here](https://docs.google.com/a/google.com/document/d/1F8IJ_7fMHZ75XFPocMokLxVZczAhrBRBVN9uMhQFCZ4/edit#heading=h.e2ug3mq1m700). Furthermore, [until recently](https://groups.google.com/d/msg/ledger-cli/A4-_OL3vvGE/Qn_4Qd5z9msJ) Ledger was not using cost basis to balance postings in a way that correctly computes capital gains. For this reason I suspect Ledger users have probably not used it to compute and compare their gains against the values reported by their broker.

In order for Beancount to detect such errors meaningfully and implement a strict matching discipline when reducing lots, it turns out that the only constraint it needs to apply is that a particular account not be allowed to simultaneously hold long and short positions of the same commodity. For example, all it has to do is enforce that you could not hold a lot of 1 GOOG units and a lot of -1 GOOG units simultaneously in the same inventory (regardless of their acquisition cost). Any reduction of a particular set of units is detected as a “lot reduction” and a search is found for a lot that matches the specification of the reducing posting, normally, a search for a lot held at the same cost as the posting specifies. Enforcing this constraint is not of much concern in practice, as there are no cases where you would want both long and short positions in the same account, but if you ever needed this, you could simply resort to using separate accounts to hold your long and short positions (it is already recommended that you use a sub-account to track your positions in any one commodity anyway, this would be quite natural).

Finally, Beancount is implementing [a proposal](http://furius.ca/beancount/doc/proposal-booking) that significantly extends the scope of its inventory booking: the syntax will allow a loose specification for lot reductions that makes it easy for users to write postings where there is no ambiguity, as well as supporting booking lots at their average cost as is common in Canada and in all tax-deferred accounts across the world. It will also provide a new syntax for specifying cost per unit and total cost at the same time, a feature which will make it possible to correctly [track capital gains without commissions](http://furius.ca/beancount/doc/trading).

## Currency Conversions

Beancount makes an important semantic distinction between simple currency conversions and conversions of commodities to be held at cost, i.e., for which we want to track the cost basis. For example, converting 20,000 USD to 22,000 CAD is a currency conversion (e.g., between banks), and after inserting the resulting CAD units in the destination account, they are not considered “CAD at a cost basis of 1.1 USD each,” they are simply left as CAD units in the account, with no record of the rate which was used to convert them. This models accurately how the real world operates. On the other hand, converting 5,000 USD into 10 units of GOOG shares with a cost basis of 500 USD each is treated as a distinct operation from a currency conversion: the resulting GOOG units have a particular cost basis attached to them, a memory of the price per unit is kept on the inventory lot (as well as the date) and strict rules are applied to the reduction of such lots (what I call “booking”) as mentioned previously. This is used to calculate and report capital gains, and most importantly, it detects a *lot* of errors in data entry.

In contrast, Ledger [does not distinguish between these two types of conversions](https://groups.google.com/d/msg/ledger-cli/zRSMle5AV3Q/kBCYjVEB-gsJ). Ledger treats currency conversions the same way as for the conversion of commodities held at cost. The reason that this is not causing as many headaches as one might intuit, is because there is no inventory booking - all lots at whichever conversion rate they occur accumulate in accounts, with positive or negative values - and for simple commodities (e.g. currencies, such as dollars), netting the total amount of units without considering the cost of each unit provides the correct answer. Inspecting the full list of an account’s inventory lots is provided as an option (See Ledger’s “--lot-dates”)and is not the default way to render account balances. I suspect few users make use of the feature: if you did render the list of lots for real-world accounts in which many currency conversions occurred in the past, you would observe a large number of irrelevant lots. I think the cost basis of currency conversions would be best elided instead.

HLedger [does not parse cost basis syntax](https://groups.google.com/d/msg/hledger/ZB8pzWJGFy8/RziOVQ2XQ-QJ) and as such does not recognize it.

## Isolation of Inputs

Beancount reads its entire input *only* from the text file you provide for it. This isolation is by design. There is no linkage to external data formats nor online services, such as fetchers for historical price values. Fetching and converting external data is disparate enough that I feel it should be the province of separate projects. These problem domains also segment very well and quite naturally: Beancount provides an isolated core which allows you to ingest all the transactional data and derive reports of various aggregations from it, and its syntax is the hinge that connects it to external transaction repositories or price databases. It isolates itself from the ugly details of external sources of data in this way.

There are too many external formats for downloadable files that contains transactional information to be able to cover all of them. The data files you can obtain from most institutions are provided in various formats: OFX, Quicken, XLS or CSV, and looking at their contents makes it glaringly obvious that the programmers who built the codes that outputs them did not pay much attention to detail or the standard definition of their format; it’s quite an ugly affair. Those files are almost always very messy, and the details of that messiness varies over time as well as these files evolve.

Fetching historical or current price information is a similarly annoying task. While Yahoo and Google Finance are able to provide some basic level of price data for common stocks on US exchanges, when you need to fetch information for instruments traded on foreign exchanges, or instruments typically not traded on exchanges, such as mutual funds, either the data is not available, or if it is, you need to figure out what ticker symbol they decided to map it to, there are few standards for this. You must manually sign the ticker. Finally, it is entirely possible that you want to manage instruments for which there is no existing external price source, so it is necessary that your bookkeeping software provide a mechanism whereby you can manually input price values (both Beancount and Ledger provide a way). The same declaration mechanism is used for caching historical price data, so that Beancount need not require the network at all.

Most users will want to write their own scripts for import, but some libraries exist: the [beancount.ingest](http://bitbucket.org/blais/beancount/src/tip/beancount/ingest) library (within Beancount) provides a framework to automate the identification, extraction of transactions from and filing of downloadable files for various institutions. See its [design doc](http://furius.ca/ledgerhub/doc/design-doc) for details.

In comparison, Ledger and HLedger support [rudimentary conversions of transactions from CSV files](http://ledger-cli.org/3.0/doc/ledger3.html#Converting-from-other-formats) as well as [automated fetching of current prices](http://ledger-cli.org/3.0/doc/ledger3.html#Commodity-reporting) that uses an external script you are meant to provide (getquote). CSV import is far insufficient for real world usage if you are to track all your accounts, so this needs to be extended. Hooking into an external script is the right thing to do, but Beancount favors taking a strong stance about this issue and instead not to provide code that would trigger any kind of network access nor support any external format as input. In Beancount you are meant to integrate price updates on your own, perhaps with your own script, and maybe bring in the data via an include file. (But if you don't like this, you could also write your own plugin module that could fetch live prices.)

## Language Syntax

Beancount’s syntax is somewhat simpler and quite a bit more restrictive than Ledger’s. For its 2.0 version, the Beancount syntax was redesigned to be easily specifiable to a parser generator by a grammar. The tokens were simplified in order to make tokenization unambiguous. For example,

* Currencies must be entirely in capital letters (allowing numbers and some special characters, like “\_” or “-”). Currency symbols (such as $ or €) are not supported. (On the other hand, currencies with numbers require quoting in Ledger.)
* Account names do not admit spaces (though you can use dashes), and must have at least two components, separated by colons.
* Description strings must be quoted, like this: “AMEX PMNT”. No freestyle text as strings is supported anymore.
* Dates are only parsed from ISO8601 format, that is, “YYYY-MM-DD”.
* Tags must begin with “#”, and links with “^”.
* Apart from the tag stack, all context information has been removed. There is no account alias, for example, nor is there a notion of “apply” as in Ledger (see “apply root” and “apply tag”). It requires a bit more verbose input—full account names—and so assumes that you have account name completion setup in your editor.

As a side effect, these changes make the input syntax look a bit more like a programming language. These restrictions may annoy some users, but overall they make the task of parsing the contents of a ledger simpler and the resulting simplicity will pave the way for people to more easily write parsers in other languages. (Some subtleties remain around parsing indentation, which is meaningful in the syntax, but should be easily addressable in all contexts by building a custom lexer.)

Due to its looser and more user-friendly syntax, Ledger uses a custom parser. If you need to parse its contents from another language, the best approach is probably to create bindings into its source code or to use it to export the contents of a ledger to XML and then parse that (which works well). I suspect the parsing method might be reviewed in the next version of Ledger, because using a parser generator is liberating for experimentation.

## Order Independence

Beancount offers a guarantee that the ordering of directives in an input file is irrelevant to the outcome of its computations. You should be able to organize your input file and reorder any declaration as is most convenient for you without having to worry about how the software will make its calculations. Not even directives that declare accounts (“Open”) are required to appear before these accounts get used in the file. All directives are parsed and then basically stably sorted before any calculation or validation occurs. This also makes it trivial to implement inclusions of multiple files (you can just concatenate the files if you want).

In contrast, Ledger processes the amounts on its postings as it parses the input file. In terms of implementation, this has the advantage that only a single pass is needed to check all the assertions and balances, whereas in Beancount, numerous passes are made on the entire list of directives (this has not been much of a problem in Beancount, however, because even a realistically large number of transactions is rather modest for our computers; most of Beancount’s processing time is due to parsing and numerical calculations).

An unfortunate side-effect of Ledger’s method of calculation is that a user must be careful with the order in which the transactions appear in the file. This can be treacherous and difficult to understand when editing a very large input file. This difference is [particularly visible in balance assertions](https://groups.google.com/forum/#!topic/ledger-cli/vwkrPh74NFI). Ledger’s balance assertions are attached to the postings of transaction directives and calculated in file order (I call these “file assertions”). Beancount’s balance assertions are separate directives that are applied at the beginning of the date for which they are declared, regardless of their position in the file (call these “dated assertions”). I believe that dated assertions are more useful and less error-prone, as they do not depend on the ordering of declarations. On the other hand, Ledger-style file assertions naturally support balance checks on intra-day balances without having to specify the time on transactions, which is impossible with dated assertions.

For this reason, a [proposal has been written up](http://furius.ca/beancount/doc/proposal-balance) to consider implementing file assertions in Beancount (in addition to its dated assertions). This will probably be carried out as a plugin. Ledger does not support dated assertions.

## Account Types

Beancount accounts must have a particular type from one of five categories: Assets, Liabilities, Income, Expenses and Equity. Ledger accounts are not constrained in this way, you can define any root account you desire and there is no requirement to identify an account as belonging to one of these categories. This reflects the more liberal approach of Ledger: its design aims to be a more general “calculator” for anything you want. No account types are enforced or used by the system.

In my experience, I haven’t seen any case where I could not classify one of my accounts in one of those categories. For the more exotic commodities, e.g., “US dollars allowable to contribute to an IRA”, it might require a bit of imagination to understand which account fits which category, but there is a logic to it: if the account has an *absolute* value that we care about, then it is an Assets or Liabilities account; if we care only about the *transitional* values, or the value accumulated during a time period, then it should be an Income or Expenses account. If the sign is positive, then it should be an Assets or Expenses account; conversely, if the sign is negative, it should be a Liabilities or Income account. Equity accounts are almost never used explicitly, and are defined and used by Beancount itself to transfer opening balances, retained earnings and net income to the balance sheet report for a particular reporting period (any period you choose). This principle makes it easy to resolve which type an account should have. I have data from 2008 to 2014 and have been able to represent *everything* I ever wanted using these five categories. Also, I don’t think asking the user to categorize their accounts in this way is limiting in any way; it just requires a bit of foresight.

The reason for requiring these account types is that it allows us to carry out logic based on their types. We can isolate the Income and Expenses accounts and derive an income statement and a single net income value. We can then transfer retained earnings (income from before the period under consideration) and net income (income during the period) to Equity accounts and draw up a balance sheet. We can generate lists of holdings which automatically exclude income and expenses, to compute net worth, value per account, etc. In addition, we can use account types to identify external flows to a group of accounts and compute the correct return on investments for these accounts in a way that can be compared with a target allocation’s market return (note: not integrated yet, but prototyped and spec’ed out, it works). The bottom line is that having account types is a useful attribute, so we enforce that you should choose a type for each account.

The absence of account types is probably also why Ledger does not provide a balance sheet or income statement reports, only a trial balance. The advantage is an apparently looser and more permissive naming structure. But also note that requiring types does not itself cause any difference in calculations between the two systems, you can still accumulate any kind of "beans" you may want in these accounts, it is no less general. The type is only extra information that Beancount’s reporting makes use of.

## Transactions Must Balance

Beancount transactions are required to balance, period. I make no compromise in this, there is no way out. The benefit of this is that the sum of the balance amounts of the postings of any subset of transactions is always precisely *zero* (and I check for it).

Ledger allows the user two special kinds of postings:

* [Virtual postings](http://ledger-cli.org/3.0/doc/ledger3.html#Virtual-postings): These are postings input with parentheses around them, and they are *not* considered in the transaction balance’s sum, you can put any value in them without causing an error.
* [Balanced virtual postings](http://ledger-cli.org/3.0/doc/ledger3.html#Working-with-multiple-funds-and-accounts): These postings are input with square brackets around them. These are less permissive: the set of postings in square brackets is enforced to balance within itself.

The second case can be shown to be equivalent to two transactions: a transaction with the set of regular postings, and a separate transaction with only the balanced virtual ones (this causes no problem). The first case is the problematic one: in attempting to solve accounting problems, beginning users routinely revert to them as a cop-out instead of modeling their problem with the double-entry method. It is apparent that most adopters of CLI accounting systems are computer scientists and not accounting professionals, and as we are all learning how to create our charts of accounts and fill up our ledgers, we are making mistakes. Oftentimes it is truly not obvious how to solve these problems; it simply requires experience.

But the fact is that in 8 years’ worth of usage, there isn’t a single case I have come across that truly required having virtual postings. The first version of Beancount used to have support for virtual postings, but I’ve managed to remove all of them over time. I was always able to come up with a better set of accounts, or to use an imaginary currency that would allow me to track whatever I needed to track. And it has always resulted in a better solution with unexpected and sometimes elegant side-effects.

But these systems have to be easy to use, so how do we address this problem? The mailing-list is a good place to begin and ask questions, where people share information regarding how they have solved similar problems (there aren’t that many “accounting problems” per-se in the first place). I am also in the process of documenting all the solutions that I have come up with to solve my own accounting problems in the [Beancount Cookbook](http://furius.ca/beancount/doc/cookbook), basically everything I’ve learned so far; this is work in progress. I hope for this evolving document to become a helpful reference to guide others in coming up with solutions that fit the double-entry accounting framework, and to provide ample examples that will act as templates for others to replicate, to fit in their own data.

In the words of Ledger’s author:

“[*If people don't want to use them [virtual accounts], that's fine. But Ledger is not an accounting tool; it's a tool that may be used to do accounting. As such, I believe virtual accounts serve a role that others with non-accounting problems may wish to fill.*](https://groups.google.com/d/msg/ledger-cli/N8Slh2t45K0/xhgMrw04miYJ)*”*

I respectfully beg to differ. Therefore Beancount takes a more radical stance and explicitly avoids supporting virtual postings. If you feel strongly that you should need them, you should use Ledger.

## Numbers and Precision of Operations

Beancount, Ledger and HLedger all differ in how they represent their numbers internally, and in how they handle the precisionof balance checks for a transaction’s postings.

First, about how number are represented: Ledger [uses rational numbers](http://www.ledger-cli.org/3.0/doc/ledger3.html#Internal-Design) in an attempt to maintain the full precision of numbers resulting from mathematical operations. This works, but [I believe this is perhaps not the most appropriate choice](https://groups.google.com/d/msg/ledger-cli/m-TgILbfrwA/YjkmOM3LHXIJ). The great majority of the cases where operations occur involve the conversion from a number of units and a price or a cost to the total value of an account’s posted change (e.g., units x cost = total cost). Our task in representing transactional information is the replication of operations that take place mostly in institutions. These operations always involve the rounding of numbers for units and currencies (banks do apply stochastic rounding), and the *correct* numbers to be used from the perspective of these institutions, and from the perspective of the government, are indeed the *rounded* numbers themselves. It is a not a question of mathematical purity, but one of practicality, and our system should do the same that banks do. Therefore, I think that we should always post the rounded numbers to accounts. Using rational numbers is not a limitation in that sense, but we must be careful to store rounded numbers where it matters. I think the approach implemented by Ledger is to keep as much of the original precision as possible.

Beancount chooses a [decimal](https://docs.python.org/3/library/decimal.html) number representation to store the numbers parsed from the input with the same precision they are written as. This method suffers from the same problem as using rational numbers does in that the result of mathematical operations between the decimal numbers will currently be stored with their full precision (albeit in decimal). Admittedly, I have yet to apply explicit quantization where required, which would be the correct thing to do. A scheme has to be devised to infer suitable precisions for automatically quantizing the numbers after operations. The decimal representation provides natural opportunities for rounding after operations, and it is a suitable choice for this, implementations even typically provide a context for the precision to take place. Also note that it will never be required to store numbers to an infinite precision: the institutions never do it themselves.

HLedger, oddly enough, [selects “double” fractional binary representation for its prices](https://groups.google.com/d/msg/ledger-cli/m-TgILbfrwA/iZAv6IK3KuYJ). This is an unfortunate choice, a worse one than using a precise representation: fractional decimal numbers input by the user are *never* represented precisely by their corresponding binary form. So all the numbers are incorrect but “close enough” that it works overall, and the only way to display a clean final result is by rounding to a suitable number of digits at the time of rendering a report. One could argue that the large number of digits provided by a 64-bit double representation is unlikely to cause significant errors given the quantity of operations we make… but binary rounding error could potentially accumulate, and the numbers are basically all incorrectly stored internally, rounded to their closest binary relative. Given that our task is accounting, why not just represent them correctly?

Secondly, when checking that the postings of a transaction balance to zero, with all three systems it is necessary to allow for some tolerance on those amounts. This need is clear when you consider that inputting numbers in a text file implies a limited decimal representation. For example, if you’re going to multiply a number of units and a cost, say both written down with 2 fractional digits, you might end up with a number that has 4 fractional digits, and then you need to compare that result with a cash amount that would typically be entered with only 2 fractional digits. You need to allow for some looseness somehow.

The systems differ in how they choose that tolerance:

* Ledger attempts to automatically derive the precision to use for its balance checks by using recently parsed context (in file order). The precision to be used is that of the last value parsed for the particular commodity under consideration. This can be problematic: it can lead to [unnecessary side-effects between transactions which can be difficult to debug](https://groups.google.com/d/msg/ledger-cli/m-TgILbfrwA/cTHg2juqEJgJ).
* HLedger, on the other hand, uses global precision settings. [The whole file is processed first, then the precisions are derived from the most precise numbers seen in the entire input file.](https://groups.google.com/d/msg/ledger-cli/m-TgILbfrwA/SoGZDNhlDOkJ)
* At the moment, Beancount uses a [constant value](https://bitbucket.org/blais/beancount/src/c194c7fa6c15a0356e9d26b20b471f0868843c42/src/python/beancount/core/complete.py?at=default#cl-25) for the tolerance used in its [balance checking algorithm](https://bitbucket.org/blais/beancount/src/c194c7fa6c15a0356e9d26b20b471f0868843c42/src/python/beancount/ops/validation.py?at=default#cl-391) (0.005 of any unit). This is weak and should, at the very least, be commodity-dependent, if not also dependent on the particular account in which the commodity is used. Ultimately, it depends on the numbers of digits used to represent the particular postings. We have a [proposal](http://furius.ca/beancount/doc/rounding) en route to fix this.

I am planning to fix this: Beancount will eventually derive its precision using a method entirely *local* to each transaction, perhaps with a global value for defaults (this is still in the works - *Oct 2014*). Half of the most precise digit will be the tolerance. This will be derived similarly to HLedger’s method, but for each transaction separately. This will allow the user to use an arbitrary precision, simply by inserting more digits in the input. Only fractional digits will be used to derive precision. No global effect implied by transactions will be applied. No transaction should ever affect any other transaction’s balancing context. [Rounding error will be optionally accumulated to an Equity account if you want to monitor it](https://groups.google.com/d/msg/ledger-cli/m-TgILbfrwA/YjkmOM3LHXIJ).

As for automatically quantizing the numbers resulting from operations, I still need to figure out an automatic method for doing so.

## Filtering at the Transactional Level

Another area where the systems differ is in that Beancount will not support filtering at the postings level but only at the transaction level. That is, when applying filters to the input data, even filtering that applies predicates to postings, only sets of *complete* transactions will be produced. This is carried out in order to produce sets of postings whose sum balances to exactly zero. We will not ignore only some postings of a transaction. We will nevertheless allow reports to cull out arbitrary subsets of accounts once all balances have been computed.

Ledger filtering works at [either the transactional or postings level](https://groups.google.com/d/msg/ledger-cli/i2ouOXgyVbE/EyWEXqZDHmkJ). I think this is confusing. I don’t really understand why it works this way or how it actually works.

(Note that this is not a very relevant point at this moment, because I have yet to implement custom arbitrary filtering; the only filtering available at the moment are “views” you can obtain through the web interface, but I will soon provide a simple logical expression language to apply custom filters to the parsed transactions as in Ledger, as I think it’s a powerful feature. Until recently, most reports were only rendered through a web interface and it wasn’t as needed as it is now that Beancount implements console reports.)

## Extension Mechanisms

Both Beancount and Ledger provide mechanisms for writing scripts against their corpus of data. These mechanisms are all useful but different.

Ledger provides

* A custom expression language which it interprets to produce reports
* A command to export contents to XML
* A command to export contents to LISP
* A library of Python bindings which provides access to its C++ data structures
* *(… others?)*

(Note: due to its dependencies, C++ features being used and build system, I have found it difficult to build Ledger on my vanilla Ubuntu machine or Mac OS computer. Building it with the Python bindings is even more difficult. If you have the patience, the time and that’s not a problem for you, great, but if you can find a pre-packaged version I would recommend using that instead.)

Beancount [provides](http://furius.ca/beancount/doc/plugins)

* A native Python plugin system whereby you may specify lists of Python modules to import and call to filter and transform the parsed directives to implement new features;
* An easy loader function that allows you to access the internal data structures resulting from parsing and processing a Beancount ledger. This is also a native Python library.

So basically, you must write Python to extend Beancount. I’m planning to provide output to XML and SQL as well (due to the simple data structures I’m using these will be both very simple to implement). Moreover, using Beancount’s own “printer” module produces text that is guaranteed to parse back into exactly the same data structure (Beancount guarantees round-tripping of its syntax and its data structures).

One advantage is that the plugins system allows you to perform in-stream arbitrary transformations of the list of directives produced, and this is a great way to prototype new functionality and easier syntax. For example, you can allocate a special tag that will trigger some arbitrary transformation on such tagged transactions. And these modules don’t have to be integrated with Beancount: they can just live anywhere on your own PYTHONPATH, so you can experiment without having to contribute new features upstream or patch the source code.

(Note: Beancount is implemented in Python 3, so you might have to install a recent version in order to work with it, such as Python-3.4, if you don’t have it. At this point, Python 3 is becoming pretty widespread, so I don’t see this as a problem, but you might be using an older OS.)

## Automated Transactions via Plugins

Ledger provides a special syntax to [automatically insert postings on existing transactions](http://ledger-cli.org/3.0/doc/ledger3.html#Automated-Transactions), based on some matching criteria. The syntax allows the user to access some of a posting’s data, such as amount and account name. The syntax is specialized to also allow the application of transaction and posting “tags.”

Beancount allows you to do the same thing and more via its plugin extension mechanism. Plugins that you write are able to completely modify, create or even delete any object and attribute of any object in the parsed flow of transactions, allowing you to carry out as much automation and summarization as you like. This does not require a special syntax - you simply work in Python, with access to all its features - and you can piggyback your automation on top of existing syntax. Some examples are provided under beancount.plugins.\*.

There is an argument to be made, however, for the availability of a quick and easy method for specifying the most common case of just adding some postings. I’m not entirely convinced yet, but Beancount may acquire this eventually (you could easily prototype this now in a plugin file if you wanted).

## No Support for Time or Effective Dates

Beancount does not represent the intra-day time of transactions, its granularity is a day. Ledger allows you to specify the time of transactions down to seconds precision. I choose to limit its scope in the interest of simplicity, and I also think there are few use cases for supporting intra-day operations.

Note that while Beancount’s maximum resolution is one day, when it sorts the directives it will maintain the relative order of all transactions that occurs within one day, so it is possible to represent multiple transactions occurring on the same day in Beancount and still do correct inventory bookings. But I believe that if you were to do day-trading you would need a more specialized system to compute intra-day returns and do technical analysis and intraday P/L calculations. Beancount is not suited for those (a lot of other features would be needed if that was its scope).

Ledger also has support for *effective dates* which are essentially an alternative date for the transaction. Reporting features allow Ledger users to use the main date or the alternative date. I used to have this feature in Beancount and I removed it, mainly because I did not want to introduce reporting options, and to handle two dates, say a transaction date and a settlement date, I wanted to enforce that at any point in time all transactions would balance. I also never made much use of it which indicates it was probably futile. Handling postings to occur at different points in time would have created imbalances, or I would have had to come up with a solution that involved “limbo” or “transfer” accounts. I preferred to just remove the feature: in 8 years of data, I have *always* been able to fudge the dates to make everything balance. This is not a big issue.

Note that handling those split transaction and merging them together will be handled; a [proposal](https://docs.google.com/document/d/1x0qqWGRHi02ef-FtUW172SHkdJ8quOZD-Xli7r4Nl_k/) is underway.

## Documents

Beancount offers support for integrating a directory hierarchy of documents with the contents of a ledger’s chart of accounts. You can provide a directory path and Beancount will automatically find and create corresponding Document directives for filenames that begin with a date in directories that match mirror account names, and attach those documents to those accounts. And given a bit of configuration, the bean-file tool can automatically file downloaded documents to such a directory hierarchy.

Ledger’s binding of documents to transactions works by generic meta-data. Users can attach arbitrary key-value pairs to their transactions, and those can be filenames. Beyond that, there is no particular support for document organization.

## Simpler and More Strict

Finally, Beancount has a generally simpler input syntax than Ledger. There are very few command-line options—and this is on purpose, I want to localize all the input within the file—and the directive syntax is more homogeneous: all transactions begin with a date and a keyword. If the argument of simplicity appeals to you, you might prefer to work with Beancount. I feel that the number of options offered in Ledger is daunting, and I could not claim to understand all of the possible ways they might interact with each other. If this does not worry you, you might prefer to use Ledger.

It is also more strict than Ledger. Certain kinds of Beancount inputs are not valid. Any transaction in an account needs to have an open directive to initiate the account, for example (though some of these constraints can be relaxed via optional plugins). If you maintain a Beancount ledger, you can expect to have to normalize it to fix a number of common errors being reported. I view this as a good thing: it detects many potential problems and applies a number of strict constraints to its input which allows us to make reasonable assumptions later on when we process the stream of directives. If you don’t care about precision or detecting potential mistakes, Ledger will allow you to be more liberal. On the other hand if you care to produce a precise and flawless account of transactions, Beancount offers more support in its validation of your inputs.

## Web Interface

Beancount has a built-in web interface, and there is an external project called [Fava](https://github.com/beancount/fava) which significantly improves on the same theme. This is the default mode for browsing reports. I believe HLedger also has a web interface as well.

# Missing Features

Beancount generally attempts to *minimize* the number of features it provides. This is in contrast with Ledger’s implementation, which has received a substantial amount of feature addition in order to experiment with double-entry bookkeeping. There are a large number of options. This reflects a difference in approach: I believe that there is a small core of essential features to be identified and that forward progress is made when we are able to minimize and remove any feature that is not strictly necessary. My goal is to provide the smallest possible kernel of features that will allow one to carry out the full spectrum of bookkeeping activities, and to make it possible for users to extend the system to automate repetitive syntax.

But here is a list of features that Beancount does not support that are supported in Ledger, and that I find would be useful, that I think would be nice to have eventually. The list below is unlikely to be exhaustive.

## ~~Console Output~~

~~Beancount’s original implementation focused on providing a web view for all of its contents. During the 2.0 rewrite I began implementing some console/text outputs, mainly because I want to be able for reports to be exportable to share with others. I have a trial balance view (like Ledger’s “bal” report) but for now the journal view isn’t implemented.~~

~~Ledger, on the other hand, has always focused on console reports.~~

~~I’ll make all the reports in Beancount support output to text format first thing after the initial release, as I’m increasingly enjoying text reports. Use bean-query --list-formats to view current status of this.~~

## ~~Filtering Language~~

~~Beancount does not yet have a filtering language. Until recently, its web interface was the main mode for rendering reports and exploring the contents of its ledger, and it provided limited subsets of transactions in the form of “views”, e.g., per-year, per-tag, etc. Having a filtering language in particular allows one to do away with many sub-accounts. I want to simplify my chart of accounts so I need this.~~

~~I’m working on adding a simple logical expression language to do arbitrary filters on the set of Beancount transactions. This is straightforward to implement and a high priority.~~

## ~~No Meta-data~~

~~Beancount does not currently support meta-data. Ledger users routinely make liberal use of metadata. This has been identified as a powerful feature and a prototype has already been implemented. Meta-data will be supported on any directive type as well as on any posting. A dictionary of key-value pairs will be attached to each of these objects. Supported values will include strings, dates, numbers, currencies and amounts.~~

~~So far the plan is to restrict Beancount’s own code to make no specific use of meta-data, on purpose. The meta-data will be strictly made available for user-plugins and custom user scripts to make use of.~~

## ~~No Arithmetic Expressions~~

~~Beancount does not support arbitrary expression evaluation in the syntax, in the places where numbers are allowed in the input. Ledger does. I have had no use for these yet, but I have no particular reason against adding this, I just haven’t implemented it, as I don’t need it myself.~~

~~I think an implementation would be straightforward and very low risk, a simple change to the parser and there is already a callback for numbers. I think there are many legitimate uses for it.~~

## Limited Support for Unicode

Beancount support UTF8 or and other encodings ***in strings only*** (that is, for input that is in quotes). For example, you can enter payees and narrations with non-ASCII characters, but not account names (which aren’t in quotes). Ledger supports other encodings over the entire file.

The reason for the lack of a more general encoding support in Beancount is current limitation of tokenizer tools. I’ve been using [GNU flex](https://www.gnu.org/software/flex/) to implement my lexer and it does not support arbitrary encodings. I just need to [write a better lexer](https://groups.google.com/d/msg/ledger-cli/C6GOOj8kGtQ/Jt9jMdRQIH4J) and make that work with [Bison](http://www.gnu.org/software/bison/), it’s not a difficult task. I will eventually write my own lexer manually—this has other advantages—and will write it to support Unicode (Python 3 has full support for this, so all that is required is to modify the lexer, which is one simple compilation unit). This is a relatively easy and isolated task to implement.

## No Forecasting or Periodic Transactions

Beancount has no support for generating periodic transactions for forecasting, though there is a plugin provided that implements a simplistic form of it to be used as an example for how plugins work (see beancount.plugins.forecast). Ledger supports [periodic transaction generation](http://www.ledger-cli.org/3.0/doc/ledger3.html#Periodic-Transactions).

I do want to add this to core Beancount eventually, but I want to define the semantics very clearly before I do. Updating one’s ledger is essentially the process of copying and replicating transactional data that happens somewhere else. I don’t believe that regular transactions are that “regular” in reality; in my experience, there is always some amount of small variations in real transactions that makes it impossible to automatically generate series of transactions by a generator in a way that would allow the user to forego updating them one-by-one. What it is useful for in my view, is for generating *tentative* future transactions. I feel strongly that those transactions should be limited not to straddle reconciled history, and that reconciled history should replace any kind of automatically generated transactions.

I have some fairly complete ideas for implementing this feature, but I’m not using forecasting myself at the moment, so it’s on the backburner. While in theory you can forecast using Ledger’s periodic transactions, to *precisely* represent your account history you will likely need to adjust the beginning dates of those transactions every time you append new transactions to your accounts and replaced forecasted ones. I don’t find the current semantics of automated transactions in Ledger to be very useful beyond creating an approximation of account contents.

(In the meantime, given the ease of extending Beancount with plugins, I suggest you begin experimenting with forecast transactions on your own for now, and if we can derive a generic way to create them, I’d be open to merging that into the main code.)