Ledger: Command-Line Accounting

For Version 3.0 of Ledger

John Wiegley

Table of Contents

1 Introduction to Ledger

1.1 Fat-free Accounting

Ledger is an accounting tool with the moxie to exist. It provides no bells or whistles, and returns the user to the days before user interfaces were even a twinkling in their father's CRT.

What it does offer is a double-entry accounting journal with all the flexibility and muscle of its modern day cousins, without any of the fat. Think of it as the Bran Muffin of accounting tools.

To use it, you need to start keeping a journal. This is the basis of all accounting, and if you haven't started yet, now is the time to learn. The little booklet that comes with your checkbook is a journal, so we'll describe double-entry accounting in terms of that.

A checkbook journal records debits (subtractions, or withdrawals) and credits (additions, or deposits) with reference to a single account: the checking account. Where the money comes from, and where it goes to, are described in the payee field, where you write the person or company's name. The ultimate aim of keeping a checkbook journal is to know how much money is available to spend. That's really the aim of all journals.

What computers add is the ability to walk through these postings, and tell you things about your spending habits; to let you devise budgets and get control over your spending; to squirrel away money into virtual savings account without having to physically move money around; etc. As you keep your journal, you are recording information about your life and habits, and sometimes that information can start telling you things you aren't aware of. Such is the aim of all good accounting tools.

The next step up from a checkbook journal, is a journal that keeps track of all your accounts, not just checking. In such a journal, you record not only who gets paid—in the case of a debit—but where the money came from. In a checkbook journal, it's assumed that all the money comes from your checking account. But in a general journal, you write postings in two lines: the source account and target account. *There must always be a debit from at least one account for every credit made to another account.* This is what is meant by "double-entry" accounting: the journal must always balance to zero, with an equal number of debits and credits.

For example, let's say you have a checking account and a brokerage account, and you can write checks from both of them. Rather than keep two checkbooks, you decide to use one journal for both. In this general journal you need to record a payment to Pacific Bell for your monthly phone bill, and a transfer (via check) from your brokerage account to your checking account. The Pacific Bell bill is $23.00, let's say, and you want to pay it from your checking account. In the general journal you need to say where the money came from, in addition to where it's going to. These transactions might look like this:

```
9/29        Pacific Bell                $23.00     $23.00
            Checking                   $-23.00          0
9/30        Checking                   $100.00    $100.00
      (123) Brokerage                 $-100.00          0
```

The posting must balance to $0: $23 went to Pacific Bell, $23 came from Checking. The next entry shows check number 123 written against your brokerage account, transferring

money to your checking account. There is nothing left over to be accounted for, since the money has simply moved from one account to another in both cases. This is the basis of double-entry accounting: money never pops in or out of existence; it is always a posting from one account to another.

Keeping a general journal is the same as keeping two separate journals: One for Pacific Bell and one for Checking. In that case, each time a payment is written into one, you write a corresponding withdrawal into the other. This makes it easier to write in a "running balance", since you don't have to look back at the last time the account was referenced—but it also means having a lot of journal books, if you deal with multiple accounts.

Here is a good place for an aside on the use of the word "account". Most private people consider an account to be something that holds money at an institution for them. Ledger uses a more general definition of the word. An account is anywhere money can go. Other finance programs use "categories", Ledger uses accounts. So, for example, if you buy some groceries at Trader Joe's, then more groceries at Whole Food Market, you might assign the transactions like this

```
2011/03/15   Trader Joe's
    Expenses:Groceries   $100.00
    Assets:Checking
2011/03/15   Whole Food Market
    Expenses:Groceries   $75.00
    Assets:Checking
```

In both cases the money goes to the '`Groceries`' account, even though the payees were different. You can set up your accounts in any way you choose.

Enter the beauty of computerized accounting. The purpose of the Ledger program is to make general journal accounting simple, by keeping track of the balances for you. Your only job is to enter the postings. If an individual posting does not balance, Ledger displays an error and indicates the incorrect posting.[1]

In summary, there are two aspects of Ledger use: updating the journal data file, and using the Ledger tool to view the summarized result of your transactions.

And just for the sake of example—as a starting point for those who want to dive in head-first—here are the journal transactions from above, formatted as the Ledger program wishes to see them:

```
2004/09/29 Pacific Bell
    Expenses:Pacific Bell             $23.00
    Assets:Checking
```

The account balances and registers in this file, if saved as `ledger.dat`, could be reported using:

```
$ ledger -f ledger.dat balance
             $-23.00  Assets:Checking
              $23.00  Expenses:Pacific Bell
--------------------
                   0
```

Or

```
$ ledger -f ledger.dat register checking
```

[1] In some special cases, it automatically balances this transaction for you.

```
04-Sep-29 Pacific Bell          Assets:Checking             $-23.00      $-23.00
```

And even:

```
$ ledger -f ledger.dat register Bell
04-Sep-29 Pacific Bell          Expenses:Pacific Bell        $23.00       $23.00
```

An important difference between Ledger and other finance packages is that Ledger will never alter your input file. You can create and edit that file in any way you prefer, but Ledger is only for analyzing the data, not for altering it.

1.2 Building the program

Ledger is written in ANSI C++, and should compile on any platform. It depends on the GNU multiple precision arithmetic library (libgmp), and the Perl regular expression library (libpcre). It was developed using GNU make and gcc 3.3, on a PowerBook running OS/X.

To build and install once you have these libraries on your system, enter these commands:

```
$ ./configure && make install
```

1.3 Getting help

Ledger has a complete online help system based on GNU Info. This manual can be searched directly from the command line using the following options: `ledger --help` brings up this entire manual in your TTY.

If you need help on how to use Ledger, or run into problems, you can join the Ledger mailing list at `http://groups.google.com/group/ledger-cli`.

You can also find help in the `#ledger` channel on the IRC server `irc.freenode.net`.

2 Ledger Tutorial

2.1 Start a Journal File

A journal is a record of your financial transactions and will be central to using Ledger. For now we just want to get a taste of what Ledger can do. An example journal is included with the source code distribution, called `drewr3.dat` (see Appendix A [Example Journal File], page 122). Copy it someplace convenient and open up a terminal window in that directory.

If you would rather start with your own journal right away please see Chapter 4 [Keeping a Journal], page 17.

2.2 Run a Few Reports

Please note that as a command line program, Ledger is controlled from your shell. There are several different command shells that all behave slightly differently with respect to some special characters. In particular, the "bash" shell will interpret '`$`' signs differently than ledger and they must be escaped to reach the actual program. Another example is "zsh", which will interpret '`^`' differently than ledger expects. In all cases that follow you should take that into account when entering the command line arguments as given. There are too many variations between shells to give concrete examples for each.

2.2.1 Balance Report

To find the balances of all of your accounts, run this command:

```
$ ledger -f drewr3.dat balance
```

Ledger will generate:

```
            $ -3,804.00  Assets
             $ 1,396.00    Checking
                $ 30.00      Business
            $ -5,200.00    Savings
            $ -1,000.00  Equity:Opening Balances
             $ 6,654.00  Expenses
             $ 5,500.00    Auto
                $ 20.00    Books
               $ 300.00    Escrow
               $ 334.00    Food:Groceries
               $ 500.00    Interest:Mortgage
            $ -2,030.00  Income
            $ -2,000.00    Salary
               $ -30.00    Sales
               $ -63.60  Liabilities
               $ -20.00    MasterCard
               $ 200.00    Mortgage:Principal
              $ -243.60    Tithe
--------------------
            $ -243.60
```

Showing you the balance of all accounts. Options and search terms can pare this down to show only the accounts you want.

A more useful report is to show only your Assets and Liabilities:

```
$ ledger -f drewr3.dat balance Assets Liabilities
```

```
          $ -3,804.00  Assets
           $ 1,396.00    Checking
              $ 30.00      Business
          $ -5,200.00    Savings
             $ -63.60  Liabilities
             $ -20.00    MasterCard
             $ 200.00    Mortgage:Principal
            $ -243.60    Tithe
--------------------
          $ -3,867.60
```

2.2.2 Register Report

To show all transactions and a running total:

```
$ ledger -f drewr3.dat register
```

Ledger will generate:

```
10-Dec-01 Checking balance      Assets:Checking          $ 1,000.00   $ 1,000.00
                                Equit:Opening Balances  $ -1,000.00           0
10-Dec-20 Organic Co-op         Expense:Food:Groceries      $ 37.50      $ 37.50
                                Expense:Food:Groceries      $ 37.50      $ 75.00
                                Expense:Food:Groceries      $ 37.50     $ 112.50
                                Expense:Food:Groceries      $ 37.50     $ 150.00
                                Expense:Food:Groceries      $ 37.50     $ 187.50
                                Expense:Food:Groceries      $ 37.50     $ 225.00
                                Assets:Checking           $ -225.00           0
10-Dec-28 Acme Mortgage         Lia:Mortgage:Principal     $ 200.00     $ 200.00
                                Expe:Interest:Mortgage     $ 500.00     $ 700.00
                                Expenses:Escrow            $ 300.00   $ 1,000.00
                                Assets:Checking          $ -1,000.00           0
11-Jan-02 Grocery Store         Expense:Food:Groceries      $ 65.00      $ 65.00
                                Assets:Checking            $ -65.00           0
11-Jan-05 Employer              Assets:Checking          $ 2,000.00   $ 2,000.00
                                Income:Salary           $ -2,000.00           0
                                (Liabilities:Tithe)       $ -240.00    $ -240.00
11-Jan-14 Bank                  Assets:Savings             $ 300.00     $ 60.00
                                Assets:Checking           $ -300.00    $ -240.00
11-Jan-19 Grocery Store         Expense:Food:Groceries      $ 44.00    $ -196.00
                                Assets:Checking            $ -44.00    $ -240.00
11-Jan-25 Bank                  Assets:Checking          $ 5,500.00   $ 5,260.00
                                Assets:Savings          $ -5,500.00    $ -240.00
11-Jan-25 Tom's Used Cars       Expenses:Auto            $ 5,500.00   $ 5,260.00
                                Assets:Checking         $ -5,500.00    $ -240.00
11-Jan-27 Book Store            Expenses:Books              $ 20.00    $ -220.00
                                Liabilities:MasterCard     $ -20.00    $ -240.00
11-Dec-01 Sale                  Asse:Checking:Business      $ 30.00    $ -210.00
                                Income:Sales               $ -30.00    $ -240.00
                                (Liabilities:Tithe)         $ -3.60    $ -243.60
```

To limit this to a more useful subset, simply add the accounts you are interested in seeing transactions for:

```
$ ledger -f drewr3.dat register Groceries
10-Dec-20 Organic Co-op         Expense:Food:Groceries      $ 37.50      $ 37.50
                                Expense:Food:Groceries      $ 37.50      $ 75.00
                                Expense:Food:Groceries      $ 37.50     $ 112.50
                                Expense:Food:Groceries      $ 37.50     $ 150.00
                                Expense:Food:Groceries      $ 37.50     $ 187.50
                                Expense:Food:Groceries      $ 37.50     $ 225.00
```

```
11-Jan-02 Grocery Store         Expense:Food:Groceries      $ 65.00     $ 290.00
11-Jan-19 Grocery Store         Expense:Food:Groceries      $ 44.00     $ 334.00
```

Which matches the balance reported for the '`Groceries`' account:

```
$ ledger -f drewr3.dat balance Groceries
            $ 334.00  Expenses:Food:Groceries
```

If you would like to find transaction to only a certain payee use '`payee`' or '`@`':

```
$ ledger -f drewr3.dat register payee "Organic"
10-Dec-20 Organic Co-op         Expense:Food:Groceries      $ 37.50      $ 37.50
                                Expense:Food:Groceries      $ 37.50      $ 75.00
                                Expense:Food:Groceries      $ 37.50     $ 112.50
                                Expense:Food:Groceries      $ 37.50     $ 150.00
                                Expense:Food:Groceries      $ 37.50     $ 187.50
                                Expense:Food:Groceries      $ 37.50     $ 225.00
                                Assets:Checking           $ -225.00           0
```

2.2.3 Cleared Report

A very useful report is to show what your obligations are versus what expenditures have actually been recorded. It can take several days for a check to clear, but you should treat it as money spent. The `cleared` report shows just that (note that the `cleared` report will not format correctly for accounts that contain multiple commodities):

```
$ ledger -f drewr3.dat cleared
     $ -3,804.00           $ 775.00                Assets
      $ 1,396.00           $ 775.00    10-Dec-20     Checking
         $ 30.00                  0                    Business
     $ -5,200.00                  0                  Savings
     $ -1,000.00        $ -1,000.00    10-Dec-01    Equity:Opening Balances
      $ 6,654.00           $ 225.00                Expenses
      $ 5,500.00                  0                  Auto
         $ 20.00                  0                  Books
        $ 300.00                  0                  Escrow
        $ 334.00           $ 225.00    10-Dec-20     Food:Groceries
        $ 500.00                  0                  Interest:Mortgage
     $ -2,030.00                  0                Income
     $ -2,000.00                  0                  Salary
        $ -30.00                  0                  Sales
        $ -63.60                  0                Liabilities
        $ -20.00                  0                  MasterCard
        $ 200.00                  0                  Mortgage:Principal
       $ -243.60                  0                  Tithe
---------------  ----------------    ---------
       $ -243.60                  0
```

The first column shows the outstanding balance, the second column shows the "cleared" balance.

2.2.4 Using the Windows Command Line

Using ledger under the windows command shell has one significant limitation. CMD.EXE is limited to standard ASCII characters and as such cannot display any currency symbols other than dollar signs '`$`'.

3 Principles of Accounting with Ledger

3.1 Accounting with Ledger

Accounting is simply tracking your money. It can range from nothing, and just waiting for automatic overdraft protection to kick in, or not, to a full-blown double-entry accounting system. Ledger accomplishes the latter. With ledger you can handle your personal finances or your business's. Double-entry accounting scales.

3.2 Stating where money goes

Accountants will talk of "credits" and "debits", but the meaning is often different from the layman's understanding. To avoid confusion, Ledger uses only subtractions and additions, although the underlying intent is the same as standard accounting principles.

Recall that every posting will involve two or more accounts. Money is transferred from one or more accounts to one or more other accounts. To record the posting, an amount is *subtracted* from the source accounts, and *added* to the target accounts.

In order to write a Ledger transaction correctly, you must determine where the money comes from and where it goes to. For example, when you are paid a salary, you must add money to your bank account and also subtract it from an income account:

```
9/29  My Employer
    Assets:Checking                          $500.00
    Income:Salary                           $-500.00
```

Why is the Income a negative figure? When you look at the balance totals for your ledger, you may be surprised to see that Expenses are a positive figure, and Income is a negative figure. It may take some getting used to, but to properly use a general ledger you must think in terms of how money moves. Rather than Ledger "fixing" the minus signs, let's understand why they are there.

When you earn money, the money has to come from somewhere. Let's call that somewhere "society". In order for society to give you an income, you must take money away (withdraw) from society in order to put it into (make a payment to) your bank. When you then spend that money, it leaves your bank account (a withdrawal) and goes back to society (a payment). This is why Income will appear negative—it reflects the money you have drawn from society—and why Expenses will be positive—it is the amount you've given back. These additions and subtractions will always cancel each other out in the end, because you don't have the ability to create new money: it must always come from somewhere, and in the end must always leave. This is the beginning of economy, after which the explanation gets terribly difficult.

Based on that explanation, here's another way to look at your balance report: every negative figure means that that account or person or place has less money now than when you started your ledger; and every positive figure means that that account or person or place has more money now than when you started your ledger. Make sense?

3.3 Assets and Liabilities

Assets are money that you have, and Liabilities are money that you owe. "Liabilities" is just a more inclusive name for Debts.

An Asset is typically increased by transferring money from an Income account, such as when you get paid. Here is a typical transaction:

```
2004/09/29  My Employer
    Assets:Checking               $500.00
    Income:Salary
```

Money, here, comes from an Income account belonging to '`My Employer`', and is transferred to your checking account. The money is now yours, which makes it an Asset.

Liabilities track money owed to others. This can happen when you borrow money to buy something, or if you owe someone money. Here is an example of increasing a MasterCard liability by spending money with it:

```
2004/09/30  Restaurant
    Expenses:Dining                $25.00
    Liabilities:MasterCard
```

The Dining account balance now shows $25 spent on Dining, and a corresponding $25 owed on the MasterCard—and therefore shown as $-25.00. The MasterCard liability shows up as negative because it offsets the value of your assets.

The combined total of your Assets and Liabilities is your net worth. So to see your current net worth, use this command:

```
$ ledger balance ^assets ^liabilities
             $500.00  Assets:Checking
             $-25.00  Liabilities:MasterCard
--------------------
             $475.00
```

In a similar vein, your Income accounts show up negative, because they transfer money *from* an account in order to increase your assets. Your Expenses show up positive because that is where the money went to. The combined total of Income and Expenses is your cash flow. A positive cash flow means you are spending more than you make, since income is always a negative figure. To see your current cash flow, use this command:

```
$ ledger balance ^income ^expenses
              $25.00  Expenses:Dining
            $-500.00  Income:Salary
--------------------
            $-475.00
```

Another common question to ask of your expenses is: How much do I spend each month on X? Ledger provides a simple way of displaying monthly totals for any account. Here is an example that summarizes your monthly automobile expenses:

```
$ ledger -M register -f drewr3.dat expenses:auto
11-Jan-01 - 11-Jan-31           Expenses:Auto            $ 5,500.00   $ 5,500.00
```

This assumes, of course, that you use account names like '`Expenses:Auto:Gas`' and '`Expenses:Auto:Repair`'.

3.3.1 Tracking reimbursable expenses

Sometimes you will want to spend money on behalf of someone else, which will eventually get repaid. Since the money is still *yours*, it is really an asset. And since the expenditure was for someone else, you don't want it contaminating your Expenses reports. You will need to keep an account for tracking reimbursements.

This is fairly easy to do in ledger. When spending the money, spend it *to* your Assets:Reimbursements, using a different account for each person or business that you spend money for. For example:

```
2004/09/29  Circuit City
    Assets:Reimbursements:Company XYZ     $100.00
    Liabilities:MasterCard
```

This shows $100.00 spent on a MasterCard at Circuit City, with the expense was made on behalf of Company XYZ. Later, when Company XYZ pays the amount back, the money will transfer from that reimbursement account back to a regular asset account:

```
2004/09/29  Company XYZ
    Assets:Checking                       $100.00
    Assets:Reimbursements:Company XYZ
```

This deposits the money owed from Company XYZ into a checking account, presumably because they paid the amount back with a check.

But what to do if you run your own business, and you want to keep track of expenses made on your own behalf, while still tracking everything in a single ledger file? This is more complex, because you need to track two separate things: 1) The fact that the money should be reimbursed to you, and 2) What the expense account was, so that you can later determine where your company is spending its money.

This kind of posting is best handled with mirrored postings in two different files, one for your personal accounts, and one for your company accounts. But keeping them in one file involves the same kinds of postings, so those are what is shown here. First, the personal transaction, which shows the need for reimbursement:

```
2004/09/29  Circuit City
    Assets:Reimbursements:Company XYZ     $100.00
    Liabilities:MasterCard
```

This is the same as above, except that you own Company XYZ, and are keeping track of its expenses in the same ledger file. This transaction should be immediately followed by an equivalent transaction, which shows the kind of expense, and also notes the fact that $100.00 is now payable to you:

```
2004/09/29  Circuit City
    Company XYZ:Expenses:Computer:Software      $100.00
    Company XYZ:Accounts Payable:Your Name
```

This second transaction shows that Company XYZ has just spent $100.00 on software, and that this $100.00 came from Your Name, which must be paid back.

These two transactions can also be merged, to make things a little clearer. Note that all amounts must be specified now:

```
2004/09/29  Circuit City
    Assets:Reimbursements:Company XYZ         $100.00
    Liabilities:MasterCard                   $-100.00
    Company XYZ:Expenses:Computer:Software    $100.00
    Company XYZ:Accounts Payable:Your Name   $-100.00
```

To "pay back" the reimbursement, just reverse the order of everything, except this time drawing the money from a company asset, paying it to accounts payable, and then drawing it again from the reimbursement account, and paying it to your personal asset account. It's easier shown than said:

```
2004/10/15  Company XYZ
    Assets:Checking                            $100.00
    Assets:Reimbursements:Company XYZ         $-100.00
    Company XYZ:Accounts Payable:Your Name     $100.00
    Company XYZ:Assets:Checking               $-100.00
```

And now the reimbursements account is paid off, accounts payable is paid off, and $100.00 has been effectively transferred from the company's checking account to your personal checking account. The money simply "waited"—in both '`Assets:Reimbursements:Company XYZ`', and '`Company XYZ:Accounts Payable:Your Name`'—until such time as it could be paid off.

The value of tracking expenses from both sides like that is that you do not contaminate your personal expense report with expenses made on behalf of others, while at the same time making it possible to generate accurate reports of your company's expenditures. It is more verbose than just paying for things with your personal assets, but it gives you a very accurate information trail.

The advantage to keep these doubled transactions together is that they always stay in sync. The advantage to keeping them apart is that it clarifies the transfer's point of view. To keep the postings in separate files, just separate the two transactions that were joined above. For example, for both the expense and the pay-back shown above, the following four transactions would be created. Two in your personal ledger file:

```
2004/09/29  Circuit City
    Assets:Reimbursements:Company XYZ     $100.00
    Liabilities:MasterCard               $-100.00

2004/10/15  Company XYZ
    Assets:Checking                        $100.00
    Assets:Reimbursements:Company XYZ     $-100.00
```

And two in your company ledger file:

```
apply account Company XYZ

2004/09/29  Circuit City
    Expenses:Computer:Software             $100.00
    Accounts Payable:Your Name            $-100.00

2004/10/15  Company XYZ
    Accounts Payable:Your Name             $100.00
    Assets:Checking                       $-100.00

end apply account
```

(Note: The `apply account` above means that all accounts mentioned in the file are children of that account. In this case it means that all activity in the file relates to Company XYZ).

After creating these transactions, you will always know that $100.00 was spent using your MasterCard on behalf of Company XYZ, and that Company XYZ spent the money on computer software and paid it back about two weeks later.

```
$ ledger balance --no-total
             $100.00  Assets:Checking
                   0  Company XYZ
            $-100.00    Assets:Checking
             $100.00    Expenses:Computer:Software
            $-100.00  Liabilities:MasterCard
```

3.4 Commodities and Currencies

Ledger makes no assumptions about the commodities you use; it only requires that you specify a commodity. The commodity may be any non-numeric string that does not contain a period, comma, forward slash or at-sign. It may appear before or after the amount, although it is assumed that symbols appearing before the amount refer to currencies, while non-joined symbols appearing after the amount refer to commodities. Here are some valid currency and commodity specifiers:

```
$20.00          ; currency: twenty US dollars
40 AAPL         ; commodity: 40 shares of Apple stock
60 DM           ; currency: 60 Deutsch Mark
£50              ; currency: 50 British pounds
50 EUR          ; currency: 50 Euros (or use appropriate symbol)
```

Ledger will examine the first use of any commodity to determine how that commodity should be printed on reports. It pays attention to whether the name of commodity was separated from the amount, whether it came before or after, the precision used in specifying the amount, whether thousand marks were used, etc. This is done so that printing the commodity looks the same as the way you use it.

An account may contain multiple commodities, in which case it will have separate totals for each. For example, if your brokerage account contains both cash, gold, and several stock quantities, the balance might look like:

```
  $200.00
100.00 AU
  AAPL 40
 BORL 100
 FEQTX 50  Assets:Brokerage
```

This balance report shows how much of each commodity is in your brokerage account.

Sometimes, you will want to know the current street value of your balance, and not the commodity totals. For this to happen, you must specify what the current price is for each commodity. The price can be any commodity, in which case the balance will be computed in terms of that commodity. The usual way to specify prices is with a price history file, which might look like this:

```
P 2004/06/21 02:18:01 FEQTX $22.49
P 2004/06/21 02:18:01 BORL $6.20
P 2004/06/21 02:18:02 AAPL $32.91
P 2004/06/21 02:18:02 AU $400.00
```

Specify the price history to use with the `--price-db` *FILE* option, with the `--market (-V)` option to report in terms of current market value:

```
$ ledger --price-db prices.db -V balance brokerage
```

The balance for your brokerage account will be reported in US dollars, since the prices database uses that currency.

```
$40880.00  Assets:Brokerage
```

You can convert from any commodity to any other commodity. Let's say you had $5000 in your checking account, and for whatever reason you wanted to know many ounces of gold that would buy, in terms of the current price of gold:

```
$ ledger -T "{1 AU}*(O/P{1 AU})" balance checking
```

Although the total expression appears complex, it is simply saying that the reported total should be in multiples of AU units, where the quantity is the account total divided by

the price of one AU. Without the initial multiplication, the reported total would still use the dollars commodity, since multiplying or dividing amounts always keeps the left value's commodity. The result of this command might be:

```
14.01 AU  Assets:Checking
```

3.4.1 Commodity price histories

Whenever a commodity is purchased using a different commodity (such as a share of common stock using dollars), it establishes a price for that commodity on that day. It is also possible, by recording price details in a ledger file, to specify other prices for commodities at any given time. Such price transactions might look like those below:

```
P 2004/06/21 02:17:58 TWCUX $27.76
P 2004/06/21 02:17:59 AGTHX $25.41
P 2004/06/21 02:18:00 OPTFX $39.31
P 2004/06/21 02:18:01 FEQTX $22.49
P 2004/06/21 02:18:02 AAPL $32.91
```

By default, ledger will not consider commodity prices when generating its various reports. It will always report balances in terms of the commodity total, rather than the current value of those commodities. To enable pricing reports, use one of the commodity reporting options.

3.4.2 Commodity equivalencies

Sometimes a commodity has several forms which are all equivalent. An example of this is time. Whether tracked in terms of minutes, hours or days, it should be possible to convert between the various forms. Doing this requires the use of commodity equivalencies.

For example, you might have the following two postings, one which transfers an hour of time into a '`Billable`' account, and another which decreases the same account by ten minutes. The resulting report will indicate that fifty minutes remain:

```
2005/10/01 Work done for company
    Billable:Client                 1h
    Project:XYZ

2005/10/02 Return ten minutes to the project
    Project:XYZ                    10m
    Billable:Client
```

Reporting the balance for this ledger file produces:

```
$ ledger --no-total balance Billable Project
               50.0m  Billable:Client
              -50.0m  Project:XYZ
```

This example works because ledger already knows how to handle seconds, minutes and hours, as part of its time tracking support. Defining other equivalencies is simple. The following is an example that creates data equivalencies, helpful for tracking bytes, kilobytes, megabytes, and more:

```
C 1.00 Kb = 1024 b
C 1.00 Mb = 1024 Kb
C 1.00 Gb = 1024 Mb
C 1.00 Tb = 1024 Gb
```

Each of these definitions correlates a commodity (such as '`Kb`') and a default precision, with a certain quantity of another commodity. In the above example, kilobytes are reported with two decimal places of precision and each kilobyte is equal to 1024 bytes.

Equivalency chains can be as long as desired. Whenever a commodity would report as a decimal amount (less than '`1.00`'), the next smallest commodity is used. If a commodity could be reported in terms of a higher commodity without resulting to a partial fraction, then the larger commodity is used.

3.5 Accounts and Inventories

Since Ledger's accounts and commodity system is so flexible, you can have accounts that don't really exist, and use commodities that no one else recognizes. For example, let's say you are buying and selling various items in EverQuest, and want to keep track of them using a ledger. Just add items of whatever quantity you wish into your EverQuest account:

```
9/29  Get some stuff at the Inn
    Places:Black's Tavern                  -3 Apples
    Places:Black's Tavern                  -5 Steaks
    EverQuest:Inventory
```

Now your EverQuest:Inventory has 3 apples and 5 steaks in it. The amounts are negative, because you are taking *from* Black's Tavern in order to add to your Inventory account. Note that you don't have to use '`Places:Black's Tavern`' as the source account. You could use '`EverQuest:System`' to represent the fact that you acquired them online. The only purpose for choosing one kind of source account over another is to generate more informative reports later on. The more you know, the better the analysis you can perform.

If you later sell some of these items to another player, the transaction would look like:

```
10/2  Sturm Brightblade
    EverQuest:Inventory                    -2 Steaks
    EverQuest:Inventory                    15 Gold
```

Now you've turned 2 steaks into 15 gold, courtesy of your customer, Sturm Brightblade.

```
$ ledger balance EverQuest
            3 Apples
             15 Gold
            3 Steaks  EverQuest:Inventory
```

3.6 Understanding Equity

The most confusing transaction in any ledger will be your equity account—because starting balances can't come out of nowhere.

When you first start your ledger, you will likely already have money in some of your accounts. Let's say there's $100 in your checking account; then add a transaction to your ledger to reflect this amount. Where will the money come from? The answer: your equity.

```
10/2  Opening Balance
    Assets:Checking                        $100.00
    Equity:Opening Balances
```

But what is equity? You may have heard of equity when people talked about house mortgages, as "the part of the house that you own". Basically, equity is like the value of something. If you own a car worth $5000, then you have $5000 in equity in that car. In order to turn that car (a commodity) into a cash flow, or a credit to your bank account, you will have to debit the equity by selling it.

When you start a ledger, you are probably already worth something. Your net worth is your current equity. By transferring the money in the ledger from your equity to your

bank accounts, you are crediting the ledger account based on your prior equity. That is why, when you look at the balance report, you will see a large negative number for Equity that never changes: Because that is what you were worth (what you debited from yourself in order to start the ledger) before the money started moving around. If the total positive value of your assets is greater than the absolute value of your starting equity, it means you are making money.

Clear as mud? Keep thinking about it. Until you figure it out, put `not Equity` at the end of your balance command, to remove the confusing figure from the total.

3.7 Dealing with Petty Cash

Something that stops many people from keeping a ledger at all is the insanity of tracking small cash expenses. They rarely generate a receipt, and there are often a lot of small postings, rather than a few large ones, as with checks.

One solution is: don't bother. Move your spending to a debit card, but in general ignore cash. Once you withdraw it from the ATM, mark it as already spent to an '`Expenses:Cash`' category:

```
2004/03/15 ATM
    Expenses:Cash                    $100.00
    Assets:Checking
```

If at some point you make a large cash expense that you want to track, just *move* the amount of the expense from '`Expenses:Cash`' into the target account:

```
2004/03/20 Somebody
    Expenses:Food                     $65.00
    Expenses:Cash
```

This way, you can still track large cash expenses, while ignoring all of the smaller ones.

3.8 Working with multiple funds and accounts

There are situations when the accounts you're tracking are different between your clients and the financial institutions where money is kept. An example of this is working as the treasurer for a religious institution. From the secular point of view, you might be working with three different accounts:

- Checking
- Savings
- Credit Card

From a religious point of view, the community expects to divide its resources into multiple "funds", from which it makes purchases or reserves resources for later:

- School fund
- Building fund
- Community fund

The problem with this kind of setup is that, when you spend money, it comes from two or more places at once: the account and the fund. And yet, the correlation of amounts between funds and accounts is rarely one-to-one. What if the school fund has '`$500.00`', but '`$400.00`' of that comes from Checking, and '`$100.00`' from Savings?

Traditional finance packages require that the money reside in only one place. But there are really two "views" of the data: from the account point of view and from the fund point of view—yet both sets should reflect the same overall expenses and cash flow. It's simply where the money resides that differs.

This situation can be handled one of two ways. The first is using virtual postings to represent the fact that money is moving to and from two kind of accounts at the same time:

```
2004/03/20 Contributions
    Assets:Checking                    $500.00
    Income:Donations

2004/03/25 Distribution of donations
    [Funds:School]                     $300.00
    [Funds:Building]                   $200.00
    [Assets:Checking]                 $-500.00
```

The use of square brackets in the second transaction ensures that the virtual postings balance to zero. Now money can be spent directly from a fund at the same time as money is drawn from a physical account:

```
2004/03/25 Payment for books (paid from Checking)
    Expenses:Books                     $100.00
    Assets:Checking                   $-100.00
    (Funds:School)                    $-100.00
```

When reports are generated, by default they'll appear in terms of the funds. In this case, you will likely want to mask out your '`Assets`' account, because otherwise the balance won't make much sense:

```
$ ledger --no-total bal not ^Assets
             $100.00  Expenses:Books
             $400.00  Funds
             $200.00    Building
             $200.00    School
            $-500.00  Income:Donations
```

If the `--real` option is used, the report will be in terms of the real accounts:

```
$ ledger --real --no-total bal
             $400.00  Assets:Checking
             $100.00  Expenses:Books
            $-500.00  Income:Donations
```

If more asset accounts are needed as the source of a posting, just list them as you would normally, for example:

```
2004/03/25 Payment for books (paid from Checking)
    Expenses:Books                     $100.00
    Assets:Checking                    $-50.00
    Liabilities:Credit Card            $-50.00
    (Funds:School)                    $-100.00
```

The second way of tracking funds is to use transaction codes. In this respect the codes become like virtual accounts that embrace the entire set of postings. Basically, we are associating a transaction with a fund by setting its code. Here are two transactions that deposit money into, and spend money from, the '`Funds:School`' fund:

```
2004/03/25 (Funds:School) Donations
    Assets:Checking                    $100.00
    Income:Donations
```

```
2004/03/25 (Funds:Building) Donations
    Assets:Checking                  $20.00
    Income:Donations

2004/04/25 (Funds:School) Payment for books
    Expenses:Books                    $50.00
    Assets:Checking
```

Note how the accounts now relate only to the real accounts, and any balance or register reports will reflect this. That the transactions relate to a particular fund is kept only in the code.

How does this become a fund report? By using the `--payee=code` option, you can generate a register report where the payee for each posting shows the code. Alone, this is not terribly interesting; but when combined with the `--by-payee (-P)` option, you will now see account subtotals for any postings related to a specific fund. So, to see the current monetary balances of all funds, the command would be:

```
$ ledger --payee=code -P reg ^Assets
04-Mar-25 Funds:Building        Assets:Checking              $20.00       $20.00
04-Mar-25 Funds:School          Assets:Checking              $50.00       $70.00
```

Or to see a particular fund's expenses, the '`School`' fund in this case:

```
$ ledger --payee=code -P reg ^Expenses and code School
04-Apr-25 Funds:School          Expenses:Books               $50.00       $50.00
```

Both approaches yield different kinds of flexibility, depending on how you prefer to think of your funds: as virtual accounts, or as tags associated with particular transactions. Your own tastes will decide which is best for your situation.

4 Keeping a Journal

The most important part of accounting is keeping a good journal. If you have a good journal, tools can be written to work whatever mathematical tricks you need to better understand your spending patterns. Without a good journal, no tool, however smart, can help you.

The Ledger program aims at making journal transactions as simple as possible. Since it is a command-line tool, it does not provide a user interface for keeping a journal. If you like, you may use GnuCash to maintain your journal, in which case Ledger will read GnuCash's data files directly. In that case, read the GnuCash manual now, and skip to the next chapter.

If you are not using GnuCash, but a text editor to maintain your journal, read on. Ledger has been designed to make data transactions as simple as possible, by keeping the journal format easy, and also by automagically determining as much information as possible based on the nature of your transactions.

For example, you do not need to tell Ledger about the accounts you use. Any time Ledger sees a posting involving an account it knows nothing about, it will create it[1]. If you use a commodity that is new to Ledger, it will create that commodity, and determine its display characteristics (placement of the symbol before or after the amount, display precision, etc.) based on how you used the commodity in the posting.

4.1 The Most Basic Entry

Here is the Pacific Bell example from above, given as a Ledger posting, with the addition of a check number:

```
9/29 (1023) Pacific Bell
    Expenses:Utilities:Phone                $23.00
    Assets:Checking                        $-23.00
```

As you can see, it is very similar to what would be written on paper, minus the computed balance totals, and adding in account names that work better with Ledger's scheme of things. In fact, since Ledger is smart about many things, you don't need to specify the balanced amount, if it is the same as the first line:

```
9/29 (1023) Pacific Bell
    Expenses:Utilities:Phone                $23.00
    Assets:Checking
```

For this transaction, Ledger will figure out that $-23.00 must come from '`Assets:Checking`' in order to balance the transaction.

Also note the structure of the account entries. There is an implied hierarchy established by separating with colons (see Section 4.3 [Structuring your Accounts], page 18).

The format is very flexible and it isn't necessary that you indent and space out things exactly as shown. The only requirements are that the start of the transaction (the date typically) is at the beginning of the first line of the transaction, and the accounts are indented by at least one space. If you omit the leading spaces in the account lines Ledger will generate an error. There must be at least two spaces, or a tab, between the amount and the account. If you do not have adequate separation between the amount and the account Ledger will give an error and stop calculating.

[1] This also means if you misspell an account it will end up getting counted separately from what you intended. The provided Emacs major mode provides for automatically filling in account names.

4.2 Starting up

Unless you have recently arrived from another planet, you already have a financial state. You need to capture that financial state so that Ledger has a starting point.

At some convenient point in time you knew the balances and outstanding obligation of every financial account you have. Those amounts form the basis of the opening entry for ledger. For example if you chose the beginning of 2011 as the date to start tracking finances with ledger, your opening balance entry could look like this:

```
2011/01/01 * Opening Balance
    Assets:Joint Checking                   $800.14
    Assets:Other Checking                    $63.44
    Assets:Savings                         $2805.54
    Assets:Investments:401K:Deferred         100.0000 VIFSX @ $80.5227
    Assets:Investments:401K:Matching          50.0000 VIFSX @ $83.7015
    Assets:Investments:IRA                   250.0000 VTHRX @ $20.5324
    Liabilities:Mortgage                 $-175634.88
    Liabilities:Car Loan                   $-3494.26
    Liabilities:Visa                       -$1762.44
    Equity:Opening Balances
```

There is nothing special about the name "Opening Balances" as the payee of the account name, anything convenient that you understand will work.

4.3 Structuring your Accounts

There really are no requirements for how you do this, but to preserve your sanity we suggest some very basic structure to your accounting system.

At the highest level you have five sorts of accounts:

1. Expenses: where money goes,
2. Assets: where money sits,
3. Income: where money comes from,
4. Liabilities: money you owe,
5. Equity: the real value of your property.

Starting the structure off this way will make it simpler for you to get answers to the questions you really need to ask about your finances.

Beneath these top level accounts you can have any level of detail you desire. For example, if you want to keep specific track of how much you spend on burgers and fries, you could have the following:

```
Expenses:Food:Hamburgers and Fries
```

4.4 Commenting on your Journal

Comments are generally started using a '`;`'. However, in order to increase compatibility with other text manipulation programs and methods, four additional comment characters are valid if used at the beginning of a line: '`#`', '`|`', and '`*`' and '`%`'.

Block comments can be made by use `comment ... end comment`.

```
; This is a single line comment,
#  and this,
%   and this,
```

```
|    and this,
*     and this.

comment
    This is a block comment with
    multiple lines
end comment
```

There are several forms of comments within a transaction, for example:

```
; this is a global comment that is not applied to a specific transaction
; it can start with any of the five characters but is not included in the
; output from 'print' or 'output'

2011/12/11  Something Sweet
    ; German Chocolate Cake
    ; :Broke Diet:
    Expenses:Food                  $10.00 ; Friends: The gang
    Assets:Credit Union:Checking
```

The first comment is global and Ledger will not attach it to any specific transactions. The comments within the transaction must all start with '`;`' and are preserved as part of the transaction. The '`:`' indicates meta-data and tags (see Section 5.7 [Metadata], page 34).

4.5 Currency and Commodities

Ledger is agnostic when it comes to how you value your accounts. Dollars, Euros, Pounds, Francs, Shares etc. are all just "commodities". Holdings in stocks, bonds, mutual funds and other financial instruments can be labeled using whatever is convenient for you (stock ticker symbols are suggested for publicly traded assets).[2]

For the rest of this manual, we will only use the word "commodities" when referring to the units on a transaction value.

This is fundamentally different than many common accounting packages, which assume the same currency throughout all of your accounts. This means if you typically operate in Euros, but travel to the US and have some expenses, you would have to do the currency conversion *before* you made the entry into your financial system. With ledger this is not required. In the same journal you can have entries in any or all commodities you actually hold. You can use the reporting capabilities to convert all commodities to a single commodity for reporting purposes without ever changing the underlying entry.

For example, the following entries reflect transactions made for a business trip to Europe from the US:

```
2011/09/23 Cash in Munich
    Assets:Cash                              €50.00
    Assets:Checking                         $-66.00

2011/09/24 Dinner in Munich
    Expenses:Business:Travel                 €35.00
    Assets:Cash
```

This says that $66.00 came out of checking and turned into 50 Euros. The implied exchange rate was $1.32. Then 35.00 Euros were spent on Dinner in Munich.

Running a ledger balance report shows:

[2] You can track *anything*, even time or distance traveled. As long as it cannot be created or destroyed inside your accounting system.

```
$ ledger -f example.dat bal
             $-66.00
              €15.00  Assets
              €15.00    Cash
             $-66.00    Checking
              €35.00  Expenses:Business:Travel
--------------------
             $-66.00
              €50.00
```

The top two lines show my current assets as $-66.00 in checking (in this very short example I didn't establish opening an opening balance for the checking account) and €15.00. After spending on dinner I have €15.00 in my wallet. The bottom line balances to zero, but is shown in two lines since we haven't told ledger to convert commodities.

4.5.1 Naming Commodities

Commodity names can have any character, including white-space. However, if you include white-space or numeric characters, the commodity name must be enclosed in double quotes '"':

```
1999/06/09 ! Achat
    Actif:SG PEE STK         49.957 "Arcancia Équilibre 454"
    Actif:SG PEE STK      $-234.90

2000/12/08 ! Achat
    Actif:SG PEE STK        215.796 "Arcancia Équilibre 455"
    Actif:SG PEE STK    $-10742.54
```

4.5.2 Buying and Selling Stock

Buying stock is a typical example that many will use that involves multiple commodities in the same transaction. The type of the share (AAPL for Apple Inc.) and the share purchase price in the currency unit you made the purchase in ($ for AAPL). Yes, the typical convention is as follows:

```
2004/05/01 Stock purchase
    Assets:Broker                     50 AAPL @ $30.00
    Expenses:Broker:Commissions        $19.95
    Assets:Broker                  $-1,519.95
```

This assumes you have a brokerage account that is capable of managing both liquid and commodity assets. Now, on the day of the sale:

```
2005/08/01 Stock sale
    Assets:Broker                    -50 APPL {$30.00} @ $50.00
    Expenses:Broker:Commissions        $19.95
    Income:Capital Gains           $-1,000.00
    Assets:Broker                   $2,480.05
```

You can, of course, elide the amount of the last posting. It is there for clarity's sake.

The '`{$30.00}`' is a lot price. You can also use a lot date, '`[2004/05/01]`', or both, in case you have several lots of the same price/date and your taxation model is based on longest-held-first.

4.5.3 Fixing Lot Prices

Commodities that you keep in order to sell at a later time have a variable value that fluctuates with the market prices. Commodities that you consume should not fluctuate in

value, but stay at the lot price they were purchased at. As an extension of "lot pricing", you can fix the per-unit price of a commodity.

For example, say you buy 10 gallons of gas at $1.20. In future "value" reports, you don't want these gallons reported in terms of today's price, but rather the price when you bought it. At the same time, you also want other kinds of commodities—like stocks— reported in terms of today's price.

This is supported as follows:

```
2009/01/01 Shell
    Expenses:Gasoline             11 GAL {=$2.299}
    Assets:Checking
```

This transaction actually introduces a new commodity, '`GAL {=$2.29}`', whose market value disregards any future changes in the price of gasoline.

If you do not want price fixing, you can specify this same transaction in one of two ways, both equivalent (note the lack of the equal sign compared to the transaction above):

```
2009/01/01 Shell
    Expenses:Gasoline             11 GAL {$2.299}
    Assets:Checking

2009/01/01 Shell
    Expenses:Gasoline             11 GAL @ $2.299
    Assets:Checking
```

There is no difference in meaning between these two forms. Why do both exist, you ask? To support things like this:

```
2009/01/01 Shell
    Expenses:Gasoline             11 GAL {=$2.299} @ $2.30
    Assets:Checking
```

This transaction says that you bought 11 gallons priced at $2.299 per gallon at a *cost to you* of $2.30 per gallon. Ledger auto-generates a balance posting in this case to Equity:Capital Losses to reflect the 1.1 cent difference, which is then balanced by Assets:Checking because its amount is null.

4.5.4 Complete control over commodity pricing

Ledger allows you to have very detailed control over how your commodities are valued. You can fine tune the results given using the `--market` or `--exchange` *`COMMODITY`* options. There are now several points of interception; you can specify the valuation method:

1. on a commodity itself,
2. on a posting, via metadata (effect is largely the same as #1),
3. on an xact, which then applies to all postings in that xact,
4. on any posting via an automated transaction,
5. on a per-account basis,
6. on a per-commodity basis,
7. by changing the journal default of `market`.

Fixated pricing (such as '`{=$20}`') still plays a role in this scheme. As far as valuation goes, it's shorthand for writing '`((s,d,t -> market($20,d,t)))`'.

A valuation function receives three arguments:

`source` A string identifying the commodity whose price is being asked for (example: '`EUR`').

`date` The reference date the price should be relative.

`target` A string identifying the "target" commodity, or the commodity the returned price should be in. This argument is null if `--market` was used instead of `--exchange` *`COMMODITY`*.

The valuation function should return an amount. If you've written your function in Python, you can return something like '`Amount("$100")`'. If the function returns an explicit value, that value is always used, regardless of the commodity, the date, or the desired target commodity. For example,

```
define myfunc_seven(s, d, t) = 7 EUR
```

In order to specify a fixed price, but still valuate that price into the target commodity, use something like this:

```
define myfunc_five(s, d, t) = market(5 EUR, d, t)
```

The `value` directive sets the valuation used for all commodities used in the rest of the data stream. This is the fallback, if nothing more specific is found.

```
value myfunc_seven
```

You can set a specific valuation function on a per-commodity basis. Instead of defining a function, you can also pass a lambda.

```
commodity $
    value s, d, t -> 6 EUR
```

Each account can also provide a default valuation function for any commodities transferred to that account.

```
account Expenses:Food5
    value myfunc_five
```

The metadata field '`Value`', if found, overrides the valuation function on a transaction-wide or per-posting basis.

```
= @XACT and Food
    ; Value:: 8 EUR
    (Equity)                    $1

= @POST and Dining
    (Expenses:Food9)            $1
        ; Value:: 9 EUR
```

Lastly, you can specify the valuation function/value for any specific amount using the '`(( ))`' commodity annotation.

```
2012-03-02 KFC
    Expenses:Food2              $1 ((2 EUR))
    Assets:Cash2

2012-03-03 KFC
    Expenses:Food3              $1
        ; Value:: 3 EUR
    Assets:Cash3

2012-03-04 KFC
    ; Value:: 4 EUR
    Expenses:Food4              $1
```

```
    Assets:Cash4

2012-03-05 KFC
    Expenses:Food5              $1
    Assets:Cash5

2012-03-06 KFC
    Expenses:Food6              $1
    Assets:Cash6

2012-03-07 KFC
    Expenses:Food7               1 CAD
    Assets:Cas7

2012-03-08 XACT
    Expenses:Food8              $1
    Assets:Cash8

2012-03-09 POST
    Expenses:Dining9            $1
    Assets:Cash9
```

```
$ ledger reg -V food
12-Mar-02 KFC                  Expenses:Food2               2 EUR        2 EUR
12-Mar-03 KFC                  Expenses:Food3               3 EUR        5 EUR
12-Mar-04 KFC                  Expenses:Food4               4 EUR        9 EUR
12-Mar-05 KFC                  Expenses:Food5                  $1          $1
                                                                        9 EUR
12-Mar-06 KFC                  Expenses:Food6                  $1          $2
                                                                        9 EUR
12-Mar-07 KFC                  Expenses:Food7               1 CAD          $2
                                                                        1 CAD
                                                                        9 EUR
12-Mar-08 XACT                 Expenses:Food8                  $1          $3
                                                                        1 CAD
                                                                        9 EUR
```

4.6 Keeping it Consistent

Sometimes Ledger's flexibility can lead to difficulties. Using a freeform text editor to enter transactions makes it easy to keep the data, but also easy to enter accounts or payees inconsistently or with spelling errors.

In order to combat inconsistency you can define allowable accounts and payees. For simplicity, create a separate text file and define accounts and payees like

```
account Expenses
account Expenses:Utilities
...
```

Using the `--strict` option will cause Ledger to complain if any accounts are not previously defined:

```
$ ledger bal --strict
Warning: "FinanceData/Master.dat", line 6: Unknown account 'Liabilities:Tithe Owed'
Warning: "FinanceData/Master.dat", line 8: Unknown account 'Liabilities:Tithe Owed'
Warning: "FinanceData/Master.dat", line 15: Unknown account 'Allocation:Equities:Domestic'
```

If you have a large Ledger register already created use the `accounts` command to get started:

```
$ ledger accounts >> Accounts.dat
```

You will have to edit this file to add the `account` directive in front of every line.

4.7 Journal Format

The ledger file format is quite simple, but also very flexible. It supports many options, though typically the user can ignore most of them. They are summarized below.

4.7.1 Transactions and Comments

The initial character of each line determines what the line means, and how it should be interpreted. Allowable initial characters are:

`NUMBER` A line beginning with a number denotes a transaction. It may be followed by any number of lines, each beginning with white-space, to denote the transaction's account postings. The format of the first line is:

```
DATE[=EDATE] [*|!] [(CODE)] DESC
```

If '`*`' appears after the date (with optional effective date), it indicates the transaction is "cleared", which can mean whatever the user wants it to mean. If '`!`' appears after the date, it indicates the transaction is "pending"; i.e., tentatively cleared from the user's point of view, but not yet actually cleared. If a `CODE` appears in parentheses, it may be used to indicate a check number, or the type of the posting. Following these is the payee, or a description of the posting.

The format of each following posting is:

```
  ACCOUNT  AMOUNT  [; NOTE]
```

The `ACCOUNT` may be surrounded by parentheses if it is a virtual posting, or square brackets if it is a virtual posting that must balance. The `AMOUNT` can be followed by a per-unit posting cost, by specifying `@ AMOUNT`, or a complete posting cost with `@@ AMOUNT`. Lastly, the `NOTE` may specify an actual and/or effective date for the posting by using the syntax `[ACTUAL_DATE]` or `[=EFFECTIVE_DATE]` or `[ACTUAL_DATE=EFFECTIVE_DATE]` (see Section 5.8 [Virtual postings], page 35).

`P` Specifies a historical price for a commodity. These are usually found in a pricing history file (see the `--download (-Q)` option). The syntax is:

```
P DATE SYMBOL PRICE
```

`=` An automated transaction. A value expression must appear after the equal sign.

After this initial line there should be a set of one or more postings, just as if it were a normal transaction. If the amounts of the postings have no commodity, they will be applied as multipliers to whichever real posting is matched by the value expression (see Section 5.22 [Automated Transactions], page 42).

`~` A periodic transaction. A period expression must appear after the tilde.

After this initial line there should be a set of one or more postings, just as if it were a normal transaction.

`; # % | *` A line beginning with a semicolon, pound, percent, bar or asterisk indicates a comment, and is ignored. Comments will not be returned in a "print" response.

`indented ;`
If the semicolon is indented and occurs inside a transaction, it is parsed as a persistent note for its preceding category. These notes or tags can be used to augment the reporting and filtering capabilities of Ledger.

4.7.2 Command Directives

`beginning of line`
Command directives must occur at the beginning of a line. Use of '`!`' and '`@`' is deprecated.

`account` Pre-declare valid account names. This only has an effect if `--strict` or `--pedantic` is used (see below). The `account` directive supports several optional sub-directives, if they immediately follow the account directive and if they begin with whitespace:

```
account Expenses:Food
    note This account is all about the chicken!
    alias food
    payee ^(KFC|Popeyes)$
    check commodity == "$"
    assert commodity == "$"
    eval print("Hello!")
    default
```

The `note` sub-directive associates a textual note with the account. This can be accessed later using the `note` value expression function in any account context.

The `alias` sub-directive, which can occur multiple times, allows the alias to be used in place of the full account name anywhere that account names are allowed.

The `payee` sub-directive, which can occur multiple times, provides regexes that identify the account if that payee is encountered and an account within its transaction ends in the name "Unknown". Example:

```
2012-02-27 KFC
    Expenses:Unknown      $10.00  ; Read now as "Expenses:Food"
    Assets:Cash
```

The `check` and `assert` directives warn or raise an error (respectively) if the given value expression evaluates to false within the context of any posting.

The `eval` directive evaluates the value expression in the context of the account at the time of definition. At the moment this has little value.

The `default` directive specifies that this account should be used as the "balancing account" for any future transactions that contain only a single posting.

`apply account`
Sets the root for all accounts following this directive. Ledger supports a hierarchical tree of accounts. It may be convenient to keep two "root accounts". For example you may be tracking your personal finances and your business finances. In order to keep them separate you could preface all personal accounts with '`personal:`' and all business accounts with '`business:`'. You can easily

split out large groups of transactions without manually editing them using the account directive. For example:

```
apply account Personal
2011/11/15  Supermarket
    Expenses:Groceries      $ 50.00
    Assets:Checking
```

Would result in all postings going into '`Personal:Expenses:Groceries`' and '`Personal:Assets:Checking`' until an '`end apply account`' directive was found.

`alias` Define an alias for an account name. If you have a deeply nested tree of accounts, it may be convenient to define an alias, for example:

```
alias Dining=Expenses:Entertainment:Dining
alias Checking=Assets:Credit Union:Joint Checking Account

2011/11/28 YummyPalace
    Dining        $10.00
    Checking
```

The aliases are only in effect for transactions read in after the alias is defined and are affected by `account` directives that precede them.

```
$ ledger bal --no-total ^Exp
              $10.00  Expenses:Entertainment:Dining
```

With the option `--recursive-aliases`, aliases can refer to other aliases, the following example produces exactly the same transactions and account names as the preceding one:

```
alias Entertainment=Expenses:Entertainment
alias Dining=Entertainment:Dining
alias Checking=Assets:Credit Union:Joint Checking Account

2011/11/30 ChopChop
  Dining          $10.00
  Checking
$ ledger balance --no-total --recursive-aliases ^Exp
              $10.00  Expenses:Entertainment:Dining
```

The option `--no-aliases` completely disables alias expansion. All accounts are read verbatim as they are in the ledger file.

`assert` An assertion can throw an error if a condition is not met during Ledger's run.

```
assert <VALUE EXPRESSION BOOLEAN RESULT>
```

`bucket` Defines the default account to use for balancing transactions. Normally, each transaction has at least two postings, which must balance to zero. Ledger allows you to leave one posting with no amount and automatically balance the transaction in the posting. The `bucket` allows you to fill in all postings and automatically generate an additional posting to the bucket account balancing the transaction. If any transaction is unbalanced, it will automatically be balanced against the `bucket` account. The following example sets '`Assets:Checking`' as the bucket:

```
bucket Assets:Checking
2011/01/25 Tom's Used Cars
    Expenses:Auto                    $ 5,500.00
```

```
2011/01/27 Book Store
    Expenses:Books                    $20.00

2011/12/01 Sale
    Assets:Checking:Business         $ 30.00
```

`capture`

Directs Ledger to replace any account matching a regex with the given account. For example:

```
capture  Expenses:Deductible:Medical  Medical
```

Would cause any posting with '`Medical`' in its name to be replaced with '`Expenses:Deductible:Medical`'.

Ledger will display the mapped payees in `print` and `register` reports.

`check` A check issues a warning if a condition is not met during Ledger's run.

```
check <VALUE EXPRESSION BOOLEAN RESULT>
```

`comment` Start a block comment, closed by `end comment`.

`commodity`

Pre-declare commodity names. This only has an effect if `--strict` or `--pedantic` is used (see below).

```
commodity $
commodity CAD
```

The `commodity` directive supports several optional sub-directives, if they immediately follow the commodity directive and—if they are on successive lines—begin with whitespace:

```
commodity $
   note American Dollars
   format $1,000.00
   nomarket
   default
```

The `note` sub-directive associates a textual note with the commodity. At present this has no value other than documentation.

The `format` sub-directive gives you a way to tell Ledger how to format this commodity. In the future, using this directive will disable Ledger's observation of other ways that commodity is used, and will provide the "canonical" representation.

The `nomarket` sub-directive states that the commodity's price should never be auto-downloaded.

The `default` sub-directive marks this as the "default" commodity.

`define` Allows you to define value expressions for future use. For example:

```
define var_name=$100

2011/12/01 Test
    Expenses  (var_name*4)
    Assets
```

The posting will have a cost of $400.

`end` Closes block commands like `tag` or `comment`.

`expr`

`fixed`

A fixed block is used to set fixated prices (see Section 5.18 [Fixated prices and costs], page 40) for a series of transactions. It's purely a typing saver, for use when entering many transactions with fixated prices.

Thus, the following:

```
fixed CAD $0.90
2012-04-10 Lunch in Canada
    Assets:Wallet            -15.50 CAD
    Expenses:Food            15.50 CAD

2012-04-11 Second day Dinner in Canada
    Assets:Wallet            -25.75 CAD
    Expenses:Food            25.75 CAD
endfixed
```

is equivalent to this:

```
2012-04-10 Lunch in Canada
    Assets:Wallet            -15.50 CAD {=$0.90}
    Expenses:Food            15.50 CAD  {=$0.90}

2012-04-11 Second day Dinner in Canada
    Assets:Wallet            -25.75 CAD  {=$0.90}
    Expenses:Food            25.75 CAD   {=$0.90}
```

Note that ending a `fixed` is done differently than other directives, as `fixed` is closed with an `endfixed` (i.e., there is *no space* between `end` and `fixed`).

For the moment, users may wish to study Bug Report 789 (`http://bugs.ledger-cli.org/show_bug.cgi?id=789`) before using the `fixed` directive in production.

`include` Include the stated file as if it were part of the current file.

`payee`

The `payee` directive supports two optional sub-directives, if they immediately follow the payee directive and—if it is on a successive line—begins with white-space:

```
payee KFC
    alias KENTUCKY FRIED CHICKEN
    uuid 2a2e21d434356f886c84371eebac6e44f1337fda
```

The `alias` sub-directive provides a regex which, if it matches a parsed payee, the declared payee name is substituted:

```
2012-02-27 KENTUCKY FRIED CHICKEN  ; will be read as being 'KFC'
...
```

The `uuid` sub-directive specifies that a transaction with exactly the uuid given should have the declared payee name substituted:

```
2014-05-13 UNHELPFUL PAYEE  ; will be read as being 'KFC'
    ; UUID: 2a2e21d434356f886c84371eebac6e44f1337fda
```

Ledger will display the mapped payees in `print` and `register` reports.

`apply tag` Allows you to designate a block of transactions and assign the same tag to all. Tags can have values and may be nested.

```
apply tag hastag
apply tag nestedtag: true

2011/01/25 Tom's Used Cars
    Expenses:Auto                    $ 5,500.00
    ; :nobudget:
    Assets:Checking

2011/01/27 Book Store
    Expenses:Books                      $20.00
    Liabilities:MasterCard

end apply tag

2011/12/01 Sale
    Assets:Checking:Business            $ 30.00
    Income:Sales

end apply tag
```

is the equivalent of:

```
2011/01/25 Tom's Used Cars
    ; :hastag:
    ; nestedtag: true
    Expenses:Auto                    $ 5,500.00
    ; :nobudget:
    Assets:Checking

2011/01/27 Book Store
    ; :hastag:
    ; nestedtag: true
    Expenses:Books                      $20.00
    Liabilities:MasterCard

2011/12/01 Sale
    ; :hastag:
    Assets:Checking:Business            $ 30.00
    Income:Sales
```

`tag` Pre-declares tag names. This only has an effect if `--strict` or `--pedantic` is used (see below).

```
tag Receipt
tag CSV
```

The `tag` directive supports two optional sub-directives, if they immediately follow the tag directive and—if on a successive line—begin with whitespace:

```
tag Receipt
  check value =~ /pattern/
  assert value != "foobar"
```

The `check` and `assert` sub-directives warn or error (respectively) if the given value expression evaluates to false within the context of any use of the related tag. In such a context, "value" is bound to the value of the tag (which may be something else but a string if typed metadata is used!). Such checks or assertions are not called if no value is given.

`test` This is a synonym for `comment` and must be closed by an `end` tag.

`year` Denotes the year used for all subsequent transactions that give a date without a year. The year should appear immediately after the directive, for example: `year 2004`. This is useful at the beginning of a file, to specify the year for that file. If all transactions specify a year, however, this command has no effect.

The following single letter commands may be at the beginning of a line alone, for backwards compatibility with older Ledger versions.

`A` See `bucket`.

`Y` See `year`.

`N SYMBOL` Indicates that pricing information is to be ignored for a given symbol, nor will quotes ever be downloaded for that symbol. Useful with a home currency, such as the dollar '$'. It is recommended that these pricing options be set in the price database file, which defaults to `~/.pricedb`. The syntax for this command is:

```
N SYMBOL
```

`D AMOUNT`

Specifies the default commodity to use, by specifying an amount in the expected format. The `xact` command will use this commodity as the default when none other can be determined. This command may be used multiple times, to set the default flags for different commodities; whichever is seen last is used as the default commodity. For example, to set US dollars as the default commodity, while also setting the thousands flag and decimal flag for that commodity, use:

```
D $1,000.00
```

`C AMOUNT1 = AMOUNT2`

Specifies a commodity conversion, where the first amount is given to be equivalent to the second amount. The first amount should use the decimal precision desired during reporting:

```
C 1.00 Kb = 1024 bytes
```

`I, i, O, o, b, h`

These four relate to timeclock support, which permits Ledger to read timelog files. See timeclock's documentation for more info on the syntax of its timelog files.

4.8 Converting from other formats

There are numerous tools to help convert various formats to a Ledger file. Most banks will generate a comma separated values file that can easily be parsed into Ledger format using one of those tools. Some of the most popular tools are:

- `ledger convert download.csv`
- `hledger -f checking.csv print`
- `icsv2ledger` (`https://github.com/quentinsf/icsv2ledger`)
- `csvToLedger` (`https://github.com/tazzben/csvToLedger`)
- `CSV2Ledger` (`https://launchpad.net/csv2ledger`)

Directly pulling information from banks is outside the scope of Ledger's function.

4.9 Archiving Previous Years

After a while, your journal can get to be pretty large. While this will not slow down Ledger—it's designed to process journals very quickly—things can start to feel "messy"; and it's a universal complaint that when finances feel messy, people avoid them.

Thus, archiving the data from previous years into their own files can offer a sense of completion, and freedom from the past. But how to best accomplish this with the ledger program? There are two commands that make it very simple: `print`, and `equity`.

Let's take an example file, with data ranging from year 2000 until 2004. We want to archive years 2000 and 2001 to their own file, leaving just 2003 and 2004 in the current file. So, use `print` to output all the earlier transactions to a file called `ledger-old.dat`:

```
$ ledger -f ledger.dat -b 2000 -e 2001 print > ledger-old.dat
```

To delete older data from the current ledger file, use `print` again, this time specifying year 2002 as the starting date:

```
$ ledger -f ledger.dat -b 2002 print > x
$ mv x ledger.dat
```

However, now the current file contains *only* postings from 2002 onward, which will not yield accurate present-day balances, because the net income from previous years is no longer being tallied. To compensate for this, we must append an equity report for the old ledger at the beginning of the new one:

```
$ ledger -f ledger-old.dat equity > equity.dat
$ cat equity.dat ledger.dat > x
$ mv x ledger.dat
$ rm equity.dat
```

Now the balances reported from `ledger.dat` are identical to what they were before the data was split.

How often should you split your ledger? You never need to, if you don't want to. Even eighty years of data will not slow down ledger much, and that's just using present day hardware! Or, you can keep the previous and current year in one file, and each year before that in its own file. It's really up to you, and how you want to organize your finances. For those who also keep an accurate paper trail, it might be useful to archive the older years to their own files, then burn those files to a CD to keep with the paper records—along with any electronic statements received during the year. In the arena of organization, just keep in mind this maxim: Do whatever keeps you doing it.

5 Transactions

5.1 Basic format

The most basic form of transaction is:

```
2012-03-10 KFC
    Expenses:Food                $20.00
    Assets:Cash                 $-20.00
```

This transaction has a date, a payee or description, a target account (the first posting), and a source account (the second posting). Each posting specifies what action is taken related to that account.

A transaction can have any number of postings:

```
2012-03-10 KFC
    Expenses:Food                $20.00
    Assets:Cash                 $-10.00
    Liabilities:Credit          $-10.00
```

5.2 Eliding amounts

The first thing you can do to make things easier is elide amounts. That is, if exactly one posting has no amount specified, Ledger will infer the inverse of the other postings' amounts:

```
2012-03-10 KFC
    Expenses:Food                $20.00
    Assets:Cash                 $-10.00
    Liabilities:Credit                  ; same as specifying $-10
```

If the other postings use multiple commodities, Ledger will copy the empty posting N times and fill in the negated values of the various commodities:

```
2012-03-10 KFC
    Expenses:Food                $20.00
    Expenses:Tips                 $2.00
    Assets:Cash              EUR -10.00
    Assets:Cash              GBP -10.00
    Liabilities:Credit
```

This transaction is identical to writing:

```
2012-03-10 KFC
    Expenses:Food                $20.00
    Expenses:Tips                 $2.00
    Assets:Cash              EUR -10.00
    Assets:Cash              GBP -10.00
    Liabilities:Credit          $-22.00
    Liabilities:Credit        EUR 10.00
    Liabilities:Credit        GBP 10.00
```

5.3 Auxiliary dates

You can associate a second date with a transaction by following the primary date with an equals sign:

```
2012-03-10=2012-03-08 KFC
    Expenses:Food                $20.00
    Assets:Cash                 $-20.00
```

What this auxiliary date means is entirely up to you. The only use Ledger has for it is that if you specify `--aux-date`, then all reports and calculations (including pricing) will use the auxiliary date as if it were the primary date.

5.4 Codes

A transaction can have a textual "code". This has no meaning and is only displayed by the print command. Checking accounts often use codes like DEP, XFER, etc., as well as check numbers. This is to give you a place to put those codes:

```
2012-03-10 (#100) KFC
    Expenses:Food               $20.00
    Assets:Checking
```

5.5 Transaction state

A transaction can have a "state": cleared, pending, or uncleared. The default is uncleared. To mark a transaction cleared, put an asterisk (*) before the payee, after the date or code:

```
2012-03-10 * KFC
    Expenses:Food               $20.00
    Assets:Cash
```

To mark it pending, use a !:

```
2012-03-10 ! KFC
    Expenses:Food               $20.00
    Assets:Cash
```

What these mean is entirely up to you. The `--cleared` option limits reports to only cleared items, while `--uncleared` shows both uncleared and pending items, and `--pending` shows only pending items.

I use cleared to mean that I've reconciled the transaction with my bank statement, and pending to mean that I'm in the middle of a reconciliation.

When you clear a transaction, that's really just shorthand for clearing all of its postings. That is:

```
2012-03-10 * KFC
    Expenses:Food               $20.00
    Assets:Cash
```

Is the same as writing:

```
2012-03-10 KFC
    * Expenses:Food               $20.00
    * Assets:Cash
```

You can mark individual postings as cleared or pending, in case one "side" of the transaction has cleared, but the other hasn't yet:

```
2012-03-10 KFC
    Liabilities:Credit           $100.00
    * Assets:Checking
```

5.6 Transaction notes

After the payee, and after at least one tab or two spaces (or a space and a tab, which Ledger calls a "hard separator"), you may introduce a note about the transaction using the '`;`' character:

```
2012-03-10 * KFC                ; yum, chicken...
    Expenses:Food                $20.00
    Assets:Cash
```

Notes can also appear on the next line, so long as that line begins with whitespace:

```
2012-03-10 * KFC                ; yum, chicken...
    ; and more notes...
    Expenses:Food                $20.00
    Assets:Cash

2012-03-10 * KFC
    ; just these notes...
    Expenses:Food                $20.00
    Assets:Cash
```

A transaction's note is shared by all its postings. This becomes significant when querying for metadata (see below). To specify that a note belongs only to one posting, place it after a hard separator after the amount, or on its own line preceded by whitespace:

```
2012-03-10 * KFC
    Expenses:Food                $20.00  ; posting #1 note
    Assets:Cash
      ; posting #2 note, extra indentation is optional
```

5.7 Metadata

One of Ledger's more powerful features is the ability to associate typed metadata with postings and transactions (by which I mean all of a transaction's postings). This metadata can be queried, displayed, and used in calculations.

The are two forms of metadata: plain tags, and tag/value pairs.

5.7.1 Metadata tags

To tag an item, put any word not containing whitespace between two colons inside a comment:

```
2012-03-10 * KFC
    Expenses:Food                $20.00
    Assets:Cash
      ; :TAG:
```

You can gang up multiple tags by sharing colons:

```
2012-03-10 * KFC
    Expenses:Food                $20.00
    Assets:Cash
      ; :TAG1:TAG2:TAG3:
```

5.7.1.1 Payee metadata tag

"Payee" is a special metadata field. If set on a posting, it will be used as the payee name for that posting. This affects the `register` report, the `payees` report, and the `--by-payee` option.

This is useful when for example you deposit 4 checks at a time to the bank. On the bank statement, there is just one amount '$400', but you can specify from whom each check came from, as shown by example below:

```
2010-06-17 Sample
    Assets:Bank        $400.00
```

```
    Income:Check1     $-100.00  ; Payee: Person One
    Income:Check2     $-100.00  ; Payee: Person Two
    Income:Check3     $-100.00  ; Payee: Person Three
    Income:Check4     $-100.00  ; Payee: Person Four
```

When reporting this, it appears as:

```
10-Jun-17 Sample            Assets:Bank             $400.00     $400.00
          Person One        Income:Check1          $-100.00     $300.00
          Person Two        Income:Check2          $-100.00     $200.00
          Person Three      Income:Check3          $-100.00     $100.00
          Person Four       Income:Check4          $-100.00           0
```

This shows that they are all in the same transaction (which is why the date is not repeated), but they have different payees now.

5.7.2 Metadata values

To associate a value with a tag, use the syntax "Key: Value", where the value can be any string of characters. Whitespace is needed after the colon, and cannot appear in the Key:

```
2012-03-10 * KFC
    Expenses:Food               $20.00
    Assets:Cash
      ; MyTag: This is just a bogus value for MyTag
```

5.7.3 Typed metadata

If a metadata tag ends in ::, its value will be parsed as a value expression and stored internally as a value rather than as a string. For example, although I can specify a date textually like so:

```
2012-03-10 * KFC
    Expenses:Food               $20.00
    Assets:Cash
      ; AuxDate: 2012/02/30
```

This date is just a string, and won't be parsed as a date unless its value is used in a date-context (at which time the string is parsed into a date automatically every time it is needed as a date). If on the other hand I write this:

```
2012-03-10 * KFC
    Expenses:Food               $20.00
    Assets:Cash
      ; AuxDate:: [2012/02/30]
```

Then it is parsed as a date only once, and during parsing of the journal file, which would let me know right away that it is an invalid date.

5.8 Virtual postings

Ordinarily, the amounts of all postings in a transaction must balance to zero. This is non-negotiable. It's what double-entry accounting is all about! But there are some tricks up Ledger's sleeve...

You can use virtual accounts to transfer amounts to an account on the sly, bypassing the balancing requirement. The trick is that these postings are not considered "real", and can be removed from all reports using `--real`.

To specify a virtual account, surround the account name with parentheses:

```
2012-03-10 * KFC
    Expenses:Food                $20.00
    Assets:Cash
    (Budget:Food)               $-20.00
```

If you want, you can state that virtual postings *should* balance against one or more other virtual postings by using brackets (which look "harder") rather than parentheses:

```
2012-03-10 * KFC
    Expenses:Food                $20.00
    Assets:Cash
    [Budget:Food]               $-20.00
    [Equity:Budgets]             $20.00
```

5.9 Expression amounts

An amount is usually a numerical figure with an (optional) commodity, but it can also be any value expression. To indicate this, surround the amount expression with parentheses:

```
2012-03-10 * KFC
    Expenses:Food      ($10.00 + $20.00)  ; Ledger adds it up for you
    Assets:Cash
```

5.10 Balance verification

If at the end of a posting's amount (and after the cost too, if there is one) there is an equals sign, then Ledger will verify that the total value for that account as of that posting matches the amount specified.

There are two forms of this features: balance assertions, and balance assignments.

5.10.1 Balance assertions

A balance assertion has this general form:

```
2012-03-10 KFC
    Expenses:Food                $20.00
    Assets:Cash                 $-20.00 = $500.00
```

This simply asserts that after subtracting $20.00 from Assets:Cash, that the resulting total matches $500.00. If not, it is an error.

5.10.2 Balance assignments

A balance assignment has this form:

```
2012-03-10 KFC
    Expenses:Food                $20.00
    Assets:Cash                         = $500.00
```

This sets the amount of the second posting to whatever it would need to be for the total in '`Assets:Cash`' to be $500.00 after the posting. If the resulting amount is not $-20.00 in this case, it is an error.

5.10.3 Resetting a balance

Say your book-keeping has gotten a bit out of date, and your Ledger balance no longer matches your bank balance. You can create an adjustment transaction using balance assignments:

```
2012-03-10 Adjustment
    Assets:Cash                          = $500.00
    Equity:Adjustments
```

Since the second posting is also null, it's value will become the inverse of whatever amount is generated for the first posting.

This is the only time in ledger when more than one posting's amount may be empty—and then only because it's not truly empty, it is indirectly provided by the balance assignment's value.

5.10.4 Balancing transactions

As a consequence of all the above, consider the following transaction:

```
2012-03-10 My Broker
    [Assets:Brokerage]            = 10 AAPL
```

What this says is: set the amount of the posting to whatever value is needed so that '`Assets:Brokerage`' contains 10 AAPL. Then, because this posting must balance, ensure that its value is zero. This can only be true if Assets:Brokerage does indeed contain 10 AAPL at that point in the input file.

A balanced virtual transaction is used simply to indicate to Ledger that this is not a "real" transaction. It won't appear in any reports anyway (unless you use a register report with `--empty`).

5.11 Posting cost

When you transfer a commodity from one account to another, sometimes it gets transformed during the transaction. This happens when you spend money on gas, for example, which transforms dollars into gallons of gasoline, or dollars into stocks in a company.

In those cases, Ledger will remember the "cost" of that transaction for you, and can use it during reporting in various ways. Here's an example of a stock purchase:

```
2012-03-10 My Broker
    Assets:Brokerage              10 AAPL
    Assets:Brokerage:Cash       $-500.00
```

This is different from transferring 10 AAPL shares from one account to another, in this case you are *exchanging* one commodity for another. The resulting posting's cost is $50.00 per share.

5.12 Explicit posting costs

You can make any posting's cost explicit using the '`@`' symbol after the amount or amount expression:

```
2012-03-10 My Broker
    Assets:Brokerage              10 AAPL @ $50.00
    Assets:Brokerage:Cash       $-500.00
```

When you do this, since Ledger can now figure out the balancing amount from the first posting's cost, you can elide the other amount:

```
2012-03-10 My Broker
    Assets:Brokerage              10 AAPL @ $50.00
    Assets:Brokerage:Cash
```

5.12.1 Primary and secondary commodities

It is a general convention within Ledger that the "top" postings in a transaction contain the target accounts, while the final posting contains the source account. Whenever a commodity is exchanged like this, the commodity moved to the target account is considered "secondary", while the commodity used for purchasing and tracked in the cost is "primary".

Said another way, whenever Ledger sees a posting cost of the form `"AMOUNT @ AMOUNT"`, the commodity used in the second amount is marked "primary".

The only meaning a primary commodity has is that the `--market (-V)` flag will never convert a primary commodity into any other commodity. `--exchange` *`COMMODITY`* `(-X)` still will, however.

5.13 Posting cost expressions

Just as you can have amount expressions, you can have posting expressions:

```
2012-03-10 My Broker
    Assets:Brokerage             10 AAPL @ ($500.00 / 10)
    Assets:Brokerage:Cash
```

You can even have both:

```
2012-03-10 My Broker
    Assets:Brokerage             (5 AAPL * 2) @ ($500.00 / 10)
    Assets:Brokerage:Cash
```

5.14 Total posting costs

The cost figure following the '`@`' character specifies the *per-unit* price for the commodity being transferred. If you'd like to specify the total cost instead, use '`@@`':

```
2012-03-10 My Broker
    Assets:Brokerage             10 AAPL @@ $500.00
    Assets:Brokerage:Cash
```

Ledger reads this as if you had written:

```
2012-03-10 My Broker
    Assets:Brokerage             10 AAPL @ ($500.00 / 10)
    Assets:Brokerage:Cash
```

5.15 Virtual posting costs

Normally whenever a commodity exchange like this happens, the price of the exchange (such as $50 per share of AAPL, above) is recorded in Ledger's internal price history database. To prevent this from happening in the case of an exceptional transaction, surround the '`@`' or '`@@`' with parentheses:

```
2012-03-10 My Brother
    Assets:Brokerage           1000 AAPL (@) $1
    Income:Gifts Received
```

5.16 Commodity prices

When a transaction occurs that exchanges one commodity for another, Ledger records that commodity price not only within its internal price database, but also attached to

the commodity itself. Usually this fact remains invisible to the user, unless you turn on `--lot-prices` to show these hidden price figures.

For example, consider the stock sale given above:

```
2012-03-10 My Broker
    Assets:Brokerage             10 AAPL @ $50.00
    Assets:Brokerage:Cash
```

The commodity transferred into '`Assets:Brokerage`' is not actually 10 AAPL, but rather 10 AAPL {$5.00}. The figure in braces after the amount is called the "lot price". It's Ledger's way of remembering that this commodity was transferred through an exchange, and that $5.00 was the price of that exchange.

This becomes significant if you later sell that commodity again. For example, you might write this:

```
2012-04-10 My Broker
    Assets:Brokerage:Cash
    Assets:Brokerage            -10 AAPL @ $75.00
```

And that would be perfectly fine, but how do you track the capital gains on the sale? It could be done with a virtual posting:

```
2012-04-10 My Broker
    Assets:Brokerage:Cash
    Assets:Brokerage            -10 AAPL @ $75.00
    (Income:Capital Gains)      $-250.00
```

But this gets messy since capital gains income is very real, and not quite appropriate for a virtual posting.

Instead, if you reference that same hidden price annotation, Ledger will figure out that the price of the shares you're selling, and the cost you're selling them at, don't balance:

```
2012-04-10 My Broker
    Assets:Brokerage:Cash       $750.00
    Assets:Brokerage            -10 AAPL {$50.00} @ $75.00
```

This transaction will fail because the $250.00 price difference between the price you bought those shares at, and the cost you're selling them for, does not match. The lot price also identifies which shares you purchased on that prior date.

5.16.1 Total commodity prices

As a shorthand, you can specify the total price instead of the per-share price in doubled braces. This goes well with total costs, but is not required to be used with them:

```
2012-04-10 My Broker
    Assets:Brokerage:Cash       $750.00
    Assets:Brokerage            -10 AAPL {{$500.00}} @@ $750.00
    Income:Capital Gains       $-250.00
```

It should be noted that this is a convenience only for cases where you buy and sell whole lots. The {{$500.00}} is *not* an attribute of the commodity, whereas {$5.00} is. In fact, when you write {{$500.00}}, Ledger just divides that value by 10 and sees {$50.00}. So if you use the print command to look at this transaction, you'll see the single braces form in the output. The double braces price form is a shorthand only.

Plus, it comes with dangers. This works fine:

```
2012-04-10 My Broker
    Assets:Brokerage            10 AAPL @ $50.00
```

```
    Assets:Brokerage:Cash       $750.00

2012-04-10 My Broker
    Assets:Brokerage:Cash       $375.00
    Assets:Brokerage            -5 AAPL {$50.00} @ $375.00
    Income:Capital Gains       $-125.00

2012-04-10 My Broker
    Assets:Brokerage:Cash       $375.00
    Assets:Brokerage            -5 AAPL {$50.00} @ $375.00
    Income:Capital Gains       $-125.00
```

But this does not do what you might expect:

```
2012-04-10 My Broker
    Assets:Brokerage            10 AAPL @ $50.00
    Assets:Brokerage:Cash       $750.00

2012-04-10 My Broker
    Assets:Brokerage:Cash       $375.00
    Assets:Brokerage            -5 AAPL {{$500.00}} @ $375.00
    Income:Capital Gains       $-125.00

2012-04-10 My Broker
    Assets:Brokerage:Cash       $375.00
    Assets:Brokerage            -5 AAPL {{$500.00}} @ $375.00
    Income:Capital Gains       $-125.00
```

And in cases where the amounts do not divide into whole figures and must be rounded, the capital gains figure could be off by a cent. Use with caution.

5.17 Prices versus costs

Because lot pricing provides enough information to infer the cost, the following two transactions are equivalent:

```
2012-04-10 My Broker
    Assets:Brokerage            10 AAPL @ $50.00
    Assets:Brokerage:Cash       $750.00

2012-04-10 My Broker
    Assets:Brokerage            10 AAPL {$50.00}
    Assets:Brokerage:Cash       $750.00
```

However, note that what you see in some reports may differ, for example in the print report. Functionally, however, there is no difference, and neither the register nor the balance report are sensitive to this difference.

5.18 Fixated prices and costs

If you buy a stock last year, and ask for its value today, Ledger will consult its price database to see what the most recent price for that stock is. You can short-circuit this lookup by "fixing" the price at the time of a transaction. This is done using {=AMOUNT}:

```
2012-04-10 My Broker
    Assets:Brokerage            10 AAPL {=$50.00}
    Assets:Brokerage:Cash       $750.00
```

These 10 AAPL will now always be reported as being worth $50, no matter what else happens to the stock in the meantime.

Fixated prices are a special case of using lot valuation expressions (see below) to fix the value of a commodity lot.

Since price annotations and costs are largely interchangeable and a matter of preference, there is an equivalent syntax for specified fixated prices by way of the cost:

```
2012-04-10 My Broker
    Assets:Brokerage            10 AAPL @ =$50.00
    Assets:Brokerage:Cash       $750.00
```

This is the same as the previous transaction, with the same caveats found in Section 5.17 [Prices versus costs], page 40.

5.19 Lot dates

In addition to lot prices, you can specify lot dates and reveal them with `--lot-dates`. Other than that, however, they have no special meaning to Ledger. They are specified after the amount in square brackets (the same way that dates are parsed in value expressions):

```
2012-04-10 My Broker
    Assets:Brokerage:Cash       $375.00
    Assets:Brokerage            -5 AAPL {$50.00} [2012-04-10] @ $375.00
    Income:Capital Gains       $-125.00
```

5.20 Lot notes

You can also associate arbitrary notes for your own record keeping in parentheses, and reveal them with `--lot-notes`. One caveat is that the note cannot begin with an '`@`' character, as that would indicate a virtual cost:

```
2012-04-10 My Broker
    Assets:Brokerage:Cash       $375.00
    Assets:Brokerage            -5 AAPL {$50.00} [2012-04-10] (Oh my!) @ $375.00
    Income:Capital Gains       $-125.00
```

You can specify any combination of lot prices, dates or notes, in any order. They are all optional.

To show all lot information in a report, use `--lots`.

5.21 Lot value expressions

Normally when you ask Ledger to display the values of commodities held, it uses a value expression called "market" to determine the most recent value from its price database—even downloading prices from the Internet, if `--download (-Q)` was specified and a suitable `getquote` script is found on your system.

However, you can override this valuation logic by providing a commodity valuation expression in doubled parentheses. This expression must result in one of two values: either an amount to always be used as the per-share price for that commodity; or a function taking three arguments, which is called to determine that price.

If you use the functional form, you can either specify a function name, or a lambda expression. Here's a function that yields the price as $10 in whatever commodity is being requested:

```
define ten_dollars(s, date, t) = market($10, date, t)
```

I can now use that in a lot value expression as follows:

```
2012-04-10 My Broker
    Assets:Brokerage:Cash       $375.00
    Assets:Brokerage            -5 AAPL {$50.00} ((ten_dollars)) @@ $375.00
    Income:Capital Gains       $-125.00
```

Alternatively, I could do the same thing without pre-defining a function by using a lambda expression taking three arguments:

```
2012-04-10 My Broker
    A:B:Cash       $375.00
    A:B     -5 AAPL {$50.00} ((s, d, t -> market($10, date, t))) @@ $375.00
    Income:Capital Gains       $-125.00
```

The arguments passed to these functions have the following meaning:

- source The source commodity string, or an amount object. If it is a string, the return value must be an amount representing the price of the commodity identified by that string (example: '$'). If it is an amount, return the value of that amount as a new amount (usually calculated as commodity price times source amount).
- date The date to use for determining the value. If null, it means no date was specified, which can mean whatever you want it to mean.
- target If not null, a string representing the desired target commodity that the commodity price, or repriced amount, should be valued in. Note that this string can be a comma-separated list, and that some or all of the commodities in that list may be suffixed with an exclamation mark, to indicate what is being desired.

In most cases, it is simplest to either use explicit amounts in your valuation expressions, or just pass the arguments down to '`market`' after modifying them to suit your needs.

5.22 Automated Transactions

An automated transaction is a special kind of transaction which adds its postings to other transactions any time one of that other transactions' postings matches its predicate. The predicate uses the same query syntax as the Ledger command line.

Consider this posting:

```
2012-03-10 KFC
    Expenses:Food               $20.00
    Assets:Cash
```

If I write this automated transaction before it in the file:

```
= expr true
    Foo                         $50.00
    Bar                        $-50.00
```

Then the first transaction will be modified during parsing as if I'd written this:

```
2012-03-10 KFC
    Expenses:Food               $20.00
    Foo                         $50.00
    Bar                        $-50.00
    Assets:Cash                $-20.00
    Foo                         $50.00
    Bar                        $-50.00
```

Despite this fancy logic, automated transactions themselves follow most of the same rules as regular transactions: their postings must balance (unless you use a virtual posting), you can have metadata, etc.

One thing you cannot do, however, is elide amounts in an automated transaction.

5.22.1 Amount multipliers

As a special case, if an automated transaction's posting's amount (phew) has no commodity, it is taken as a multiplier upon the matching posting's cost. For example:

```
= expr true
    Foo                          50.00
    Bar                         -50.00

2012-03-10 KFC
    Expenses:Food               $20.00
    Assets:Cash
```

Then the latter transaction turns into this during parsing:

```
2012-03-10 KFC
    Expenses:Food                $20.00
    Foo                        $1000.00
    Bar                       $-1000.00
    Assets:Cash                 $-20.00
    Foo                        $1000.00
    Bar                       $-1000.00
```

5.22.2 Accessing the matching posting's amount

If you use an amount expression for an automated transaction's posting, that expression has access to all the details of the matched posting. For example, you can refer to that posting's amount using the "amount" value expression variable:

```
= expr true
    (Foo)                 (amount * 2)  ; same as just "2" in this case

2012-03-10 KFC
    Expenses:Food               $20.00
    Assets:Cash
```

This becomes:

```
2012-03-10 KFC
    Expenses:Food               $20.00
    (Foo)                       $40.00
    Assets:Cash                $-20.00
    (Foo)                      $-40.00
```

5.22.3 Referring to the matching posting's account

Sometimes you want to refer to the account that was matched in some way within the automated transaction itself. This is done by using the string '`$account`', anywhere within the account part of the automated posting:

```
= food
    (Budget:$account)              10

2012-03-10 KFC
    Expenses:Food               $20.00
    Assets:Cash
```

Becomes:

```
2012-03-10 KFC
    Expenses:Food               $20.00
    (Budget:Expenses:Food)     $200.00
    Assets:Cash                $-20.00
```

5.22.4 Applying metadata to every matched posting

If the automated transaction has a transaction note, that note is copied (along with any metadata) to every posting that matches the predicate:

```
= food
    ; Foo: Bar
    (Budget:$account)               10

2012-03-10 KFC
    Expenses:Food                $20.00
    Assets:Cash
```

Becomes:

```
2012-03-10 KFC
    Expenses:Food                $20.00
      ; Foo: Bar
    (Budget:Expenses:Food)      $200.00
    Assets:Cash                 $-20.00
```

5.22.5 Applying metadata to the generated posting

If the automated transaction's posting has a note, that note is carried to the generated posting within the matched transaction:

```
= food
    (Budget:$account)               10
      ; Foo: Bar

2012-03-10 KFC
    Expenses:Food                $20.00
    Assets:Cash
```

Becomes:

```
2012-03-10 KFC
    Expenses:Food                $20.00
    (Budget:Expenses:Food)      $200.00
      ; Foo: Bar
    Assets:Cash                 $-20.00
```

This is slightly different from the rules for regular transaction notes, in that an automated transaction's note does not apply to every posting within the automated transaction itself, but rather to every posting it matches.

5.22.6 State flags

Although you cannot mark an automated transaction as a whole as cleared or pending, you can mark its postings with a '`*`' or '`!`' before the account name, and that state flag gets carried to the generated posting.

5.22.7 Effective Dates

In the real world, transactions do not take place instantaneously. Purchases can take several days to post to a bank account. And you may pay ahead for something for which you want to distribute costs. With Ledger you can control every aspect of the timing of a transaction.

Say you're in business. If you bill a customer, you can enter something like

```
2008/01/01=2008/01/14 Client invoice  ;  estimated date you'll be paid
    Assets:Accounts Receivable            $100.00
```

```
    Income: Client name
```

Then, when you receive the payment, you change it to

```
2008/01/01=2008/01/15 Client invoice ;  actual date money received
    Assets:Accounts Receivable            $100.00
    Income: Client name
```

and add something like

```
2008/01/15 Client payment
    Assets:Checking                       $100.00
    Assets:Accounts Receivable
```

Now

```
$ ledger --begin 2008/01/01 --end 2008/01/14 bal Income
```

gives you your accrued income in the first two weeks of the year, and

```
$ ledger --effective --begin 2008/01/01 --end 2008/01/14 bal Income
```

gives you your cash basis income in the same two weeks.

Another use is distributing costs out in time. As an example, suppose you just prepaid into a local vegetable co-op that sustains you through the winter. It costs $225 to join the program, so you write a check. You don't want your October grocery budget to be blown because you bought food ahead, however. What you really want is for the money to be evenly distributed over the next six months so that your monthly budgets gradually take a hit for the vegetables you'll pick up from the co-op, even though you've already paid for them.

```
2008/10/16 * (2090) Bountiful Blessings Farm
    Expenses:Food:Groceries                $ 37.50  ; [=2008/10/01]
    Expenses:Food:Groceries                $ 37.50  ; [=2008/11/01]
    Expenses:Food:Groceries                $ 37.50  ; [=2008/12/01]
    Expenses:Food:Groceries                $ 37.50  ; [=2009/01/01]
    Expenses:Food:Groceries                $ 37.50  ; [=2009/02/01]
    Expenses:Food:Groceries                $ 37.50  ; [=2009/03/01]
    Assets:Checking
```

This entry accomplishes this. Every month you'll see an automatic $37.50 deficit like you should, while your checking account really knows that it debited $225 this month.

And using the `--effective` option, the initial date will be overridden by the effective dates.

```
$ ledger --effective register Groceries
```

```
08-Oct-01 Bountiful Blessings.. Expense:Food:Groceries      $ 37.50      $ 37.50
08-Nov-01 Bountiful Blessings.. Expense:Food:Groceries      $ 37.50      $ 75.00
08-Dec-01 Bountiful Blessings.. Expense:Food:Groceries      $ 37.50     $ 112.50
09-Jan-01 Bountiful Blessings.. Expense:Food:Groceries      $ 37.50     $ 150.00
09-Feb-01 Bountiful Blessings.. Expense:Food:Groceries      $ 37.50     $ 187.50
09-Mar-01 Bountiful Blessings.. Expense:Food:Groceries      $ 37.50     $ 225.00
```

5.22.8 Periodic Transactions

A periodic transaction starts with a '~' followed by a period expression. Periodic transactions are used for budgeting and forecasting only, they have no effect without the `--budget` option specified. For examples and details, see Chapter 9 [Budgeting and Forecasting], page 93.

5.22.9 Concrete Example of Automated Transactions

As a Bahá'í, I need to compute Huqúqu'lláh whenever I acquire assets. It is similar to tithing for Jews and Christians, or to Zakát for Muslims. The exact details of computing Huqúqu'lláh are somewhat complex, but if you have further interest, please consult the Web.

Ledger makes this otherwise difficult law very easy. Just set up an automated posting at the top of your ledger file:

```
; This automated transaction will compute Huqúqu'lláh based on this
; journal's postings.  Any accounts that match will affect the
; Liabilities:Huququ'llah account by 19% of the value of that posting.

= /^(?:Income:|Expenses:(?:Business|Rent$|Furnishings|Taxes|Insurance))/
  (Liabilities:Huququ'llah)              0.19
```

This automated posting works by looking at each posting in the ledger file. If any match the given value expression, 19% of the posting's value is applied to the '`Liabilities:Huququ'llah`' account. So, if $1000 is earned from '`Income:Salary`', $190 is added to '`Liabilities:Huqúqu'lláh`'; if $1000 is spent on Rent, $190 is subtracted.

```
2003/01/01 (99) Salary
  Income:Salary  -$1000
  Assets:Checking

2003/01/01 (100) Rent
  Expenses:Rent  $500
  Assets:Checking
```

The ultimate balance of Huqúqu'lláh reflects how much is owed in order to fulfill one's obligation to Huqúqu'lláh. When ready to pay, just write a check to cover the amount shown in '`Liabilities:Huququ'llah`'. That transaction would look like:

```
2003/01/01 (101) Baha'i Huqúqu'lláh Trust
    Liabilities:Huququ'llah          $1,000.00
    Assets:Checking
```

That's it. To see how much Huqúq is currently owed based on your ledger transactions, use:

```
$ ledger balance Liabilities:Huquq
                $-95  Liabilities:Huququ'llah
```

This works fine, but omits one aspect of the law: that Huqúq is only due once the liability exceeds the value of 19 mithqáls of gold (which is roughly 2.22 ounces). So what we want is for the liability to appear in the balance report only when it exceeds the present day value of 2.22 ounces of gold. This can be accomplished using the command:

```
$ ledger -Q -t "/Liab.*Huquq/?(a/P{2.22 AU}<={-1.0}&a):a" bal liab
```

With this command, the current price for gold is downloaded, and the Huqúqu'lláh is reported only if its value exceeds that of 2.22 ounces of gold. If you wish the liability to be reflected in the parent subtotal either way, use this instead:

```
$ ledger -Q -T "/Liab.*Huquq/?(O/P{2.22 AU}<={-1.0}&O):O" bal liab
```

In some cases, you may wish to refer to the account of whichever posting matched your automated transaction's value expression. To do this, use the special account name '`$account`':

```
= /^Some:Long:Account:Name/
    [$account]  -0.10
    [Savings]    0.10
```

This example causes 10% of the matching account's total to be deferred to the '`Savings`' account—as a balanced virtual posting, which may be excluded from reports by using `--real`.

6 Building Reports

6.1 Introduction

The power of Ledger comes from the incredible flexibility in its reporting commands, combined with formatting commands. Some options control what is included in the calculations, and formatting controls how it is displayed. The combinations are infinite. This chapter will show you the basics of combining various options and commands. In the next chapters you will find details about the specific commands and options.

6.2 Balance Reports

6.2.1 Controlling the Accounts and Payees

The balance report is the most commonly used report. The simplest invocation is:

```
$ ledger balance -f drewr3.dat
```

which will print the balances of every account in your journal.

```
            $ -3,804.00  Assets
             $ 1,396.00    Checking
                $ 30.00      Business
            $ -5,200.00    Savings
            $ -1,000.00  Equity:Opening Balances
             $ 6,654.00  Expenses
             $ 5,500.00    Auto
                $ 20.00    Books
               $ 300.00    Escrow
               $ 334.00    Food:Groceries
               $ 500.00    Interest:Mortgage
            $ -2,030.00  Income
            $ -2,000.00    Salary
               $ -30.00    Sales
               $ -63.60  Liabilities
               $ -20.00    MasterCard
               $ 200.00    Mortgage:Principal
              $ -243.60    Tithe
--------------------
              $ -243.60
```

Most times, this is more than you want. Limiting the results to specific accounts is as easy as entering the names of the accounts after the command:

```
$ ledger balance -f drewr3.dat Auto MasterCard
              $ 5,500.00  Expenses:Auto
                $ -20.00  Liabilities:MasterCard
--------------------
              $ 5,480.00
```

Note the implicit logical or between '`Auto`' and '`Mastercard`'.

If you want the entire contents of a branch of your account tree, use the highest common name in the branch:

```
$ ledger balance -f drewr3.dat Income
            $ -2,030.00  Income
            $ -2,000.00    Salary
               $ -30.00    Sales
```

```
--------------------
         $ -2,030.00
```

You can use general regular expressions in nearly any place Ledger needs a string:

```
$ ledger balance -f drewr3.dat ^Bo
```

This first example looks for any account starting with '`Bo`', of which there are none.

```
$ ledger balance -f drewr3.dat Bo
             $ 20.00  Expenses:Books
```

This second example looks for any account containing '`Bo`', which is '`Expenses:Books`'.

If you want to know exactly how much you have spent in a particular account on a particular payee, the following are equivalent:

```
$ ledger balance Auto:Fuel and Chevron
$ ledger balance --limit 'account=~/Fuel/' and 'payee=~/Chev/'
```

will show you the amount expended on gasoline at Chevron. The second example is the first example of the very power expression language available to shape reports. The first example may be easier to remember, but learning to use the second will open up far more possibilities.

If you want to exclude specific accounts from the report, you can exclude multiple accounts with parentheses:

```
$ ledger bal Expenses and not (Expenses:Drinks or Expenses:Candy or Expenses:Gifts)
```

6.2.2 Controlling Formatting

These examples all use the default formatting for the balance report. Customizing the formatting can easily allowing to see only what you want, or interface Ledger with other programs.

6.3 Typical queries

A query such as the following shows all expenses since last October, sorted by total:

```
$ ledger -b "last oct" -S T bal ^expenses
```

From left to right the options mean: Show transactions since last October; sort by the absolute value of the total; and report the balance for all accounts that begin with '`expenses`'.

6.3.1 Reporting monthly expenses

The following query makes it easy to see monthly expenses, with each month's expenses sorted by the amount:

```
$ ledger -M --period-sort "(amount)" reg ^expenses
```

Now, you might wonder where the money came from to pay for these things. To see that report, add `--related (-r)`, which shows the "related account" postings:

```
$ ledger -M --period-sort "(amount)" -r reg ^expenses
```

But maybe this prints too much information. You might just want to see how much you're spending with your MasterCard. That kind of query requires the use of a display predicate, since the postings calculated must match '`^expenses`', while the postings displayed must match '`mastercard`'. The command would be:

```
$ ledger -M -r --display 'account=~/mastercard/' reg ^expenses
```

This query says: Report monthly subtotals; report the "related account" postings; display only related postings whose account matches '`mastercard`', and base the calculation on postings matching '`^expenses`'.

This works just as well for reporting the overall total, too:

```
$ ledger -s -r --display "account=~/mastercard/" reg ^expenses
```

The `--subtotal (-s)` option subtotals all postings, just as `--monthly (-M)` subtotaled by the month. The running total in both cases is off, however, since a display expression is being used.

6.4 Advanced Reports

6.4.1 Asset Allocation

A very popular method of managing portfolios is to control the percent allocation of assets by certain categories. The mix of categories and the weights applied to them vary by investing philosophy, but most follow a similar pattern. Tracking asset allocation in ledger is not difficult but does require some additional effort to describe how the various assets you own contribute to the asset classes you want to track.

In our simple example we assume you want to apportion your assets into the general categories of domestic and international equities (stocks) and a combined category of bonds and cash. For illustrative purposes, we will use several publicly available mutual funds from Vanguard. The three funds we will track are the Vanguard 500 IDX FD Signal (VIFSX), the Vanguard Target Retirement 2030 (VTHRX), and the Vanguard Short Term Federal Fund (VSGBX). Each of these funds allocates assets to different categories of the investment universe and in different proportions. When you buy a share of VTHRX, that share is partially invested in equities, and partially invested in bonds and cash. Below is the asset allocation for each of the instruments listed above:

	Domestic	Global	
Symbol	Equity	Equity	bonds/cash
VIFSX	100%		
VTHRX	24.0%	56.3%	19.7%
VSGBX			100%

These numbers are available from the prospectus of any publicly available mutual fund. Of course a single stock issue is 100% equity and a single bond issue is 100% bonds.

We track purchases of specific investments using the symbol of that investment as its commodity. How do we tell Ledger that a share of VTHRX is 24% Global equity etc.? Enter automatic transactions and virtual accounts.

At the top of our ledger we enter automatic transactions that describe these proportions to Ledger. In the same entries we set up virtual accounts that let us separate these abstract calculations from our actual balances.

For the three instruments listed above, those automatic transactions would look like:

```
;
; automatic calculations for asset allocation tracking
;
= expr ( commodity == 'VIFSX' )
```

```
    (Allocation:Equities:Domestic)                1.000

= expr ( commodity == 'VTHRX' )
    (Allocation:Equities:Global)              0.240
    (Allocation:Equities:Domestic)            0.563
    (Allocation:Bonds/Cash)                   0.197

= expr ( commodity == 'VBMFX')
    (Allocation:Bonds/Cash)                   1.000
```

How do these work? First the '=' sign at the beginning of the line tells ledger this is an automatic transaction to be applied when the condition following the '=' is true. After the '=' sign is a value expression (see Chapter 11 [Value Expressions], page 96) that returns true any time a posting contains the commodity of interest.

The following line gives the proportions (not percentages) of each unit of commodity that belongs to each asset class. Whenever Ledger sees a buy or sell of a particular commodity it will credit or debit these virtual accounts with that proportion of the number of shares moved.

Now that Ledger understands how to distribute the commodities amongst the various asset classes how do we get a report that tells us our current allocation? Using the balance command and some tricky formatting!

```
ledger bal Allocation --current --format "\
        %-17((depth_spacer)+(partial_account))\
        %10(percent(market(display_total), market(parent.total)))\
        %16(market(display_total))\n%/"
```

Which yields:

```
Allocation          100.00%      $100000.00
  Bonds/Cash         38.94%       $38940.00
  Equities           61.06%       $61060.00
    Domestic         95.31%       $58196.29
    Global            4.69%        $2863.71
```

Let's look at the Ledger invocation a bit closer. The command above is split into lines for clarity. The first line is very vanilla Ledger asking for the current balances of the account in the "Allocation" tree, using a special formatter.

The magic is in the formatter. The second line simply tells Ledger to print the partial account name indented by its depth in the tree. The third line is where we calculate and display the percentages. The `display_total` command gives the values of the total calculated for the account in this line. The `parent.total` command gives the total for the next level up in the tree. `percent` formats their ratio as a percentage. The fourth line tells ledger to display the current market value of the line. The last two characters '`%/`' tell Ledger what to do for the last line, in this case, nothing.

6.4.2 Visualizing with Gnuplot

If you have the "Gnuplot" program installed, you can graph any of the above register reports. The script to do this is included in the ledger distribution, and is named `contrib/report`. Install `report` anywhere along your `PATH`, and then use `report` instead of `ledger` when doing a register report. The only thing to keep in mind is that you must specify `--amount-data (-j)` or `--total-data (-J)` to indicate whether "Gnuplot" should plot the amount, or the running total. For example, this command plots total monthly expenses made on your MasterCard.

```
$ report -j -M -r --display "account =~ /mastercard/" reg ^expenses
```

The `report` script is a very simple Bourne shell script, that passes a set of scripted commands to "Gnuplot". Feel free to modify the script to your liking, since you may prefer histograms to line plots, for example.

Here are some useful plots:

```
report -j -M reg ^expenses         # monthly expenses
report -J reg checking             # checking account balance
report -J reg ^income ^expenses    # cash flow report

# net worth report, ignoring non-$ postings

report -J -l "Ua>={\$0.01}" reg ^assets ^liab

# net worth report starting last February.  the use of a display
# predicate (-d) is needed, otherwise the balance will start at
# zero, and thus the y-axis will not reflect the true balance

report -J -l "Ua>={\$0.01}" -d "d>=[last feb]" reg ^assets ^liab
```

The last report uses both a calculation predicate `--limit` *`EXPR`* `(-l)` and a display predicate `--display` *`EXPR`* `(-d)`. The calculation predicate limits the report to postings whose amount is greater than or equal to $1 (which can only happen if the posting amount is in dollars). The display predicate limits the transactions *displayed* to just those since last February, even though those transactions from before will be computed as part of the balance.

7 Reporting Commands

7.1 Primary Financial Reports

7.1.1 The `balance` command

The `balance` command reports the current balance of all accounts. It accepts a list of optional regexes, which confine the balance report to the matching accounts. If an account contains multiple types of commodities, each commodity's total is reported separately.

7.1.2 The `equity` command

The `equity` command prints out account balances as if they were transactions. This makes it easy to establish the starting balances for an account, such as when Section 4.9 [Archiving Previous Years], page 31.

7.1.3 The `register` command

The `register` command displays all the postings occurring in a single account, line by line. The account regex must be specified as the only argument to this command. If any regexes occur after the required account name, the register will contain only those postings that match, which makes it very useful for hunting down a particular posting.

The output from `register` is very close to what a typical checkbook, or single-account ledger, would look like. It also shows a running balance. The final running balance of any register should always be the same as the current balance of that account.

If you have "Gnuplot" installed, you may plot the amount or running total of any register by using the script `report`, which is included in the Ledger distribution. The only requirement is that you add either `--amount-data (-j)` or `--total-data (-J)` to your `register` command, in order to plot either the amount or total column, respectively.

7.1.4 The `print` command

The `print` command prints out ledger transactions in a textual format that can be parsed by Ledger. They will be properly formatted, and output in the most economic form possible. The `print` command also takes a list of optional regexes, which will cause only those postings which match in some way to be printed.

The `print` command can be a handy way to clean up a ledger file whose formatting has gotten out of hand.

7.2 Reports in other Formats

7.2.1 Comma Separated Values files

7.2.1.1 The `csv` command

The `csv` command prints the desired ledger transactions in a csv format suitable for importing into other programs. You can specify the transactions to print using all the normal limiting and searching functions.

7.2.1.2 The `convert` command

The `convert` command parses a comma separated value (csv) file and prints Ledger transactions. Many banks offer csv file downloads. Unfortunately, the file formats, aside from the commas, are all different. The ledger `convert` command tries to help as much as it can.

Your bank's csv files will have fields in different orders from other banks, so there must be a way to tell Ledger what to expect. Insert a line at the beginning of the csv file that describes the fields to Ledger.

For example, this is a portion of a csv file downloaded from a credit union in the United States:

```
Account Name: VALUFIRST CHECKING
Account Number: 71
Date Range: 11/13/2011 - 12/13/2011

Transaction Number,Date,Description,Memo,Amount Debit,Amount Credit,Balance,Check Number,Fees
767718,12/13/2011,"Withdrawal","ACE HARDWARE 16335 S HOUGHTON RD",-8.80,,00001640.04,,
767406,12/13/2011,"Withdrawal","ACE HARDWARE 16335 S HOUGHTON RD",-1.03,,00001648.84,,
683342,12/13/2011,"Visa Checking","NetFlix Date 12/12/11 000326585896 5968",-21.85,,00001649.87,,
639668,12/13/2011,"Withdrawal","ID: 1741472662 CO: XXAA.COM PAYMNT",-236.65,,00001671.72,,
1113648,12/12/2011,"Withdrawal","Tuscan IT #00037657",-29.73,,00001908.37,,
```

Unfortunately, as it stands Ledger cannot read it, but you can. Ledger expects the first line to contain a description of the fields on each line of the file. The fields ledger can recognize are called `date`, `posted`, `code`, `payee` or `desc`, `amount`, `cost`, `total`, and `note`.

Delete the account description lines at the top, and replace the first line in the data above with:

```
,date,payee,note,amount,,,code,
```

Then execute ledger like this:

```
$ ledger convert download.csv --input-date-format "%m/%d/%Y"
```

Where the `--input-date-format` *`DATE_FORMAT`* option tells ledger how to interpret the dates.

Importing csv files is a lot of work, but is very amenable to scripting.

If there are columns in the bank data you would like to keep in your ledger data, besides the primary fields described above, you can name them in the field descriptor list and Ledger will include them in the transaction as meta data if it doesn't recognize the field name. For example, if you want to capture the bank transaction number and it occurs in the first column of the data use:

```
transid,date,payee,note,amount,,,code,
```

Ledger will include '`; transid: 767718`' in the first transaction is from the file above.

The `convert` command accepts three options. The most important ones are `--invert` which inverts the amount field, and `--account` *`STR`* which you can use to specify the account to balance against and `--rich-data`. When using the rich-data switch, additional metadata is stored as tags. There is, for example, a UUID field. If an entry with the same UUID tag is already included in the normal ledger file (specified via `--file` *`FILE`* `(-f)` or via the environment variable `LEDGER_FILE`) this entry will not be printed again.

You can also use `convert` with `payee` and `account` directives. First, you can use the `payee` and `alias` directive to rewrite the `payee` field based on some rules. Then you can use the account and its `payee` directive to specify the account. I use it like this, for example:

```
payee Aldi
    alias ^ALDI SUED SAGT DANKE
account Aufwand:Einkauf:Lebensmittel
    payee ^(Aldi|Alnatura|Kaufland|REWE)$
```

Note that it may be necessary for the output of '`ledger convert`' to be passed through `ledger print` a second time if you want to match on the new payee field. During the `ledger convert` run only the original payee name as specified in the csv data seems to be used.

7.2.2 The `lisp` command

The `lisp` command prints results in a form that can be read directly by Emacs Lisp. The format of the `sexp` is:

```
((BEG-POS CLEARED DATE CODE PAYEE
  (ACCOUNT AMOUNT)...)  ; list of postings
 ...)                   ; list of transactions
```

`emacs` can also be used as a synonym for `lisp`.

7.2.3 Emacs `org` Mode

The `org` command produces a journal file suitable for use in the Emacs Org mode. More details on using Org mode can be found at `http://www.orgmode.org`.

Org mode has a sub-system known as Babel which allows for literate programming. This allows you to mix text and code within the same document and automatically execute code which may generate results which will then appear in the text.

One of the languages supported by Babel is Ledger, so that you can have ledger commands embedded in a text file and have the output of ledger commands also appear in the text file. The output can be updated whenever any new ledger entries are added.

For instance, the following Org mode text document snippet illustrates a very naive but still useful application of the Babel system:

```
* A simple test of ledger in an org file
The following are some entries and I have requested that ledger be run
to generate a balance on the accounts.  I could have asked for
a register or, in fact, anything at all the ledger can do through
command line options.

#+begin_src ledger :cmdline bal :results value
2010/01/01 * Starting balance
  assets:bank:savings       £1300.00
  income:starting balances
2010/07/22 * Got paid
  assets:bank:chequing      £1000.00
  income:salary
2010/07/23 Rent
  expenses:rent              £500.00
  assets:bank:chequing
#+end_src

#+results:
:            £1800.00  assets:bank
:             £500.00    chequing
:            £1300.00    savings
:             £500.00  expenses:rent
```

```
:           £-2300.00  income
:           £-1000.00    salary
:           £-1300.00    starting balances
```

Typing `C-c C-c` anywhere in the "ledger source code block" will invoke ledger on the contents of that block and generate a "results" block. The results block can appear anywhere in the file but, by default, will appear immediately below the source code block.

You can combine multiple source code blocks before executing ledger and do all kinds of other wonderful things with Babel (and Org mode).

7.2.4 Org mode with Babel

Using Babel, it is possible to record financial transactions conveniently in an org file and subsequently generate the financial reports required.

As of Org mode 7.01, Ledger support is provided. Check the Babel documentation on Worg for instructions on how to achieve this but I currently do this directly as follows:

```
(org-babel-do-load-languages
 'org-babel-load-languages
 '((ledger . t)         ;this is the important one for this tutorial
  ))
```

Once Ledger support in Babel has been enabled, we can proceed to include Ledger entries within an org file. There are three ways (at least) in which these can be included:

1. place all Ledger entries within one single source block and execute this block with different arguments to generate the appropriate reports,
2. place Ledger entries in more than one source block and use the `noweb` literary programming approach, supported by Babel, to combine these into one block elsewhere in the file for processing by Ledger,
3. place Ledger entries in different source blocks and use `tangle` to generate a Ledger file which you can subsequently process using Ledger directly.

The first two are described in more detail in this short tutorial.

7.2.4.1 Embedded Ledger example with single source block

The easiest, albeit possibly least useful, way in which to use Ledger within an org file is to use a single source block to record all Ledger entries. The following is an example source block:

```
#+name: allinone
#+begin_src ledger
2010/01/01 * Starting balance
  assets:bank:savings       £1300.00
  income:starting balances
2010/07/22 * Got paid
  assets:bank:chequing      £1000.00
  income:salary
2010/07/23 Rent
  expenses:rent              £500.00
  assets:bank:chequing
2010/07/24 Food
  expenses:food              £150.00
  assets:bank:chequing
2010/07/31 * Interest on bank savings
  assets:bank:savings          £3.53
```

```
  income:interest
2010/07/31 * Transfer savings
  assets:bank:savings        £250.00
  assets:bank:chequing
2010/08/01 got paid again
  assets:bank:chequing      £1000.00
  income:salary
#+end_src
```

In this example, we have combined both expenses and income into one set of Ledger entries. We can now generate register and balance reports (as well as many other types of reports) using Babel to invoke Ledger with specific arguments. The arguments are passed to Ledger using the `:cmdline` header argument. In the code block above, there is no such argument so the system takes the default. For Ledger code blocks, the default `:cmdline` argument is `bal` and the result of evaluating this code block (*C-c C-c*) would be:

```
#+results: allinone()
:             £2653.53  assets:bank
:             £1100.00    chequing
:             £1553.53    savings
:              £650.00  expenses
:              £150.00    food
:              £500.00    rent
:            £-3303.53  income
:               £-3.53    interest
:            £-2000.00    salary
:            £-1300.00    starting balances
```

If, instead, you wished to generate a register of all the transactions, you would change the `#+begin_src` line for the code block to include the required command line option:

```
#+begin_src ledger :cmdline reg
```

Evaluating the code block again would generate a different report.

Having to change the actual directive on the code block and re-evaluate makes it difficult to have more than one view of your transactions and financial state. Eventually, Babel will support passing arguments to `#+call` evaluations of code blocks but this support is missing currently. Instead, we can use the concepts of literary programming, as implemented by the `noweb` features of Babel, to help us.

7.2.4.2 Multiple Ledger source blocks with `noweb`

The `noweb` feature of Babel allows us to expand references to other code blocks within a code block. For Ledger, this can be used to group transactions according to type, say, and then bring various sets of transactions together to generate reports.

Using the same transactions used above, we could consider splitting these into expenses and income, as follows:

7.2.4.3 Income Entries

The first set of entries relates to income, either monthly pay or interest, all typically going into one of my bank accounts. Here, I have placed several entries, but we could have had each entry in a separate `src` block. Note that all code blocks you wish to refer to later must have the `:noweb yes` header argument specified.

```
#+name: income
#+begin_src ledger :noweb yes
```

```
2010/01/01 * Starting balance
  assets:bank:savings       £1300.00
  income:starting balances
2010/07/22 * Got paid
  assets:bank:chequing      £1000.00
  income:salary
2010/07/31 * Interest on bank savings
  assets:bank:savings          £3.53
  income:interest
2010/07/31 * Transfer savings
  assets:bank:savings        £250.00
  assets:bank:chequing
2010/08/01 got paid again
  assets:bank:chequing      £1000.00
  income:salary
#+end_src
```

7.2.4.4 Expenses

The following entries relate to personal expenses, such as rent and food. Again, these have all been placed in a single `src` block but could have been done individually.

```
#+name: expenses
#+begin_src ledger :noweb yes
2010/07/23 Rent
  expenses:rent              £500.00
  assets:bank:chequing
2010/07/24 Food
  expenses:food              £150.00
  assets:bank:chequing
#+end_src
```

7.2.4.5 Financial Summaries

Given the ledger entries defined above in the income and expenses code blocks, we can now refer to these using the noweb expansion directives, `<<name>>`. We can now define different code blocks to generate specific reports for those transactions. Below are two examples, one to generate a balance report and one to generate a register report of all transactions.

7.2.4.6 An overall balance summary

The overall balance of your account and expenditure with a breakdown according to category is specified by passing the `:cmdline bal` argument to Ledger. This code block can now be evaluated (*C-c C-c*) and the results generated by incorporating the transactions referred to by the `<<income>>` and `<<expenses>>` lines.

```
#+name: balance
#+begin_src ledger :cmdline bal :noweb yes
<<income>>
<<expenses>>
#+end_src

#+results: balance
:            £2653.53  assets:bank
:            £1100.00    chequing
:            £1553.53    savings
:             £650.00  expenses
:             £150.00    food
```

```
:             £500.00    rent
:           £-3303.53  income
:              £-3.53    interest
:           £-2000.00    salary
:           £-1300.00    starting balances
```

If you want a less detailed breakdown of where your money is, you can specify the `--collapse (-n)` flag (i.e. '`:cmdline -n bal`') to tell Ledger to exclude sub-accounts in the report.

```
#+begin_src ledger :cmdline -n bal :noweb yes
<<income>>
<<expenses>>
#+end_src

#+results:
:            £2653.53  assets
:             £650.00  expenses
:           £-3303.53  income
```

7.2.4.7 Generating a monthly register

You can also generate a monthly register (the `reg` command) by executing the following `src` block. This presents a summary of transactions for each monthly period (the `--monthly (-M)` argument) with a running total in the final column (which should be 0 at the end if all the entries are correct).

```
#+name: monthlyregister
#+begin_src ledger :cmdline -M reg :noweb yes
<<income>>
<<expenses>>
#+end_src

#+results: monthlyregister
:2010/01/01 - 2010/01/31         assets:bank:savings       £1300.00    £1300.00
:                                in:starting balances     £-1300.00           0
:2010/07/01 - 2010/07/31         assets:bank:chequing       £100.00     £100.00
:                                assets:bank:savings        £253.53     £353.53
:                                expenses:food              £150.00     £503.53
:                                expenses:rent              £500.00    £1003.53
:                                income:interest             £-3.53    £1000.00
:                                income:salary            £-1000.00           0
:2010/08/01 - 2010/08/01         assets:bank:chequing      £1000.00    £1000.00
:                                income:salary            £-1000.00           0
```

We could also generate a monthly report on our assets showing how these are increasing (or decreasing!). In this case, the final column will be the running total of the assets in our ledger.

```
#+name: monthlyassetsregister
#+begin_src ledger :cmdline -M reg assets :noweb yes
<<income>>
<<expenses>>
#+end_src

#+results: monthlyassetsregister
: 2010/01/01 - 2010/01/31         assets:bank:savings       £1300.00    £1300.00
: 2010/07/01 - 2010/07/31         assets:bank:chequing       £100.00    £1400.00
:                                 assets:bank:savings        £253.53    £1653.53
: 2010/08/01 - 2010/08/01         assets:bank:chequing      £1000.00    £2653.53
```

7.2.4.8 Summary

This short tutorial shows how Ledger entries can be embedded in an org file and manipulated using Babel. However, only simple Ledger features have been illustrated; please refer to the Ledger documentation for examples of more complex operations on a ledger.

7.2.5 The `pricemap` command

If you have the `graphviz` graph visualization package installed, ledger can generate a graph of the relationship between your various commodities. The output file is in the "dot" format.

This is probably not very interesting, unless you have many different commodities valued in terms of each other. For example, multiple currencies and multiple investments valued in those currencies.

7.2.6 The `xml` command

By default, Ledger uses a human-readable data format, and displays its reports in a manner meant to be read on screen. For the purpose of writing tools which use Ledger, however, it is possible to read and display data using XML. This section documents that format.

The general format used for Ledger data is:

```
<?xml version="1.0"?>
<ledger>
  <xact>...</xact>
  <xact>...</xact>
  <xact>...</xact>...
</ledger>
```

The data stream is enclosed in a `ledger` tag, which contains a series of one or more transactions. Each `xact` describes one transaction and contains a series of one or more postings:

```
<xact>
  <en:date>2004/03/01</en:date>
  <en:cleared/>
  <en:code>100</en:code>
  <en:payee>John Wiegley</en:payee>
  <en:postings>
    <posting>...</posting>
    <posting>...</posting>
    <posting>...</posting>...
  </en:postings>
</xact>
```

The date format for `en:date` is always `YYYY/MM/DD`. The `en:cleared` tag is optional, and indicates whether the posting has been cleared or not. There is also an `en:pending` tag, for marking pending postings. The `en:code` and `en:payee` tags both contain whatever text the user wishes.

After the initial transaction data, there must follow a set of postings marked with `en:postings`. Typically these postings will all balance each other, but if not they will be automatically balanced into an account named '`Unknown`'.

Within the `en:postings` tag is a series of one or more `posting`'s, which have the following form:

```
<posting>
  <tr:account>Expenses:Computer:Hardware</tr:account>
```

```
  <tr:amount>
    <value type="amount">
      <amount>
        <commodity flags="PT">$</commodity>
        <quantity>90.00</quantity>
      </amount>
    </value>
  </tr:amount>
</posting>
```

This is a basic posting. It may also begin with `tr:virtual` and/or `tr:generated` tags, to indicate virtual and auto-generated postings. Then follows the `tr:account` tag, which contains the full name of the account the posting is related to. Colons separate parent from child in an account name.

Lastly follows the amount of the posting, indicated by `tr:amount`. Within this tag is a `value` tag, of which there are four different kinds, each with its own format:

1. Boolean,
2. integer,
3. amount,
4. balance.

The format of a Boolean value is `true` or `false` surrounded by a `boolean` tag, for example:

```
<boolean>true</boolean>
```

The format of an integer value is the numerical value surrounded by an `integer` tag, for example:

```
<integer>12036</integer>
```

The format of an amount contains two members, the commodity and the quantity. The commodity can have a set of flags that indicate how to display it. The meaning of the flags (all of which are optional) are:

`P` The commodity is prefixed to the value.

`S` The commodity is separated from the value by a space.

`T` Thousands markers are used to display the amount.

`E` The format of the amount is European, with period used as a thousands marker, and comma used as the decimal point.

The actual quantity for an amount is an integer of arbitrary size. Ledger uses the GNU multiple precision arithmetic library to handle such values. The XML format assumes the reader to be equally capable. Here is an example amount:

```
<value type="amount">
  <amount>
    <commodity flags="PT">$</commodity>
    <quantity>90.00</quantity>
  </amount>
</value>
```

Lastly, a balance value contains a series of amounts, each with a different commodity. Unlike the name, such a value does need to balance. It is called a balance because it sums several amounts. For example:

```
<value type="balance">
  <balance>
    <amount>
      <commodity flags="PT">$</commodity>
      <quantity>90.00</quantity>
    </amount>
    <amount>
      <commodity flags="TE">DM</commodity>
      <quantity>200.00</quantity>
    </amount>
  </balance>
</value>
```

That is the extent of the XML data format used by Ledger. It will output such data if the `xml` command is used, and can read the same data.

7.2.7 `prices` and `pricedb` commands

The `prices` command displays the price history for matching commodities. The `--average (-A)` option is useful with this report, to display the running average price, or `--deviation (-D)` to show each price's deviation from that average.

There is also a `pricedb` command which outputs the same information as `prices`, but does so in a format that can be parsed by Ledger. This is useful for generating and tidying up pricedb database files.

7.3 Reports about your Journals

7.3.1 `accounts`

The `accounts` command reports all of the accounts in the journal. Following the command with a regular expression will limit the output to accounts matching the regex. The output is sorted by name. Using the `--count` option will tell you how many entries use each account.

7.3.2 `payees`

The `payees` command reports all of the unique payees in the journal. Using the `--count` option will tell you how many entries use each payee. To filter the payees displayed you must use the prefix @:

```
$ ledger payees @Nic
Nicolas
Nicolas BOILABUS
Oudtshoorn Municipality
Vaca Veronica
```

7.3.3 `commodities`

Report all commodities present in the journals under consideration. The output is sorted by name. Using the `--count` option will tell you how many entries use each commodity.

7.3.4 `tags`

The `tags` command reports all of the tags in the journal. The output is sorted by name. Using the `--count` option will tell you how many entries use each tag. Using the `--values` option will report the values used by each tag.

7.3.5 xact

The `xact` command simplify the creation of new transactions. It works on the principle that 80% of all postings are variants of earlier postings. Here's how it works:

Say you currently have this posting in your ledger file:

```
2004/03/15 * Viva Italiano
    Expenses:Food                $12.45
    Expenses:Tips                 $2.55
    Liabilities:MasterCard       $-15.00
```

Now it's '`2004/4/9`', and you've just eating at '`Viva Italiano`' again. The exact amounts are different, but the overall form is the same. With the `xact` command you can type:

```
$ ledger xact 2004/4/9 viva food 11 tips 2.50
```

This produces the following output:

```
2004/04/09 Viva Italiano
    Expenses:Food                       $11.00
    Expenses:Tips                        $2.50
    Liabilities:MasterCard
```

It works by finding a past posting matching the regular expression '`viva`', and assuming that any accounts or amounts specified will be similar to that earlier posting. If Ledger does not succeed in generating a new transaction, an error is printed and the exit code is set to '`1`'.

Here are a few more examples of the `xact` command, assuming the above journal transaction:

```
$ ledger xact 4/9 viva 11.50
$ ledger xact 4/9 viva 11.50 checking # (from 'checking')
$ ledger xact 4/9 viva food 11.50 tips 8
$ ledger xact 4/9 viva food 11.50 tips 8 cash
$ ledger xact 4/9 viva food $11.50 tips $8 cash
$ ledger xact 4/9 viva dining "DM 11.50"
```

`draft` and `entry` are both synonyms of `xact`. `entry` is provided for backwards compatibility with Ledger 2.X.

7.3.6 stats

FIX THIS ENTRY

7.3.7 select

FIX THIS ENTRY

8 Command-line Syntax

8.1 Basic Usage

This chapter describes Ledger's features and options. You may wish to survey this to get an overview before diving into the Chapter 2 [Ledger Tutorial], page 4 and more detailed examples that follow.

Ledger has a very simple command-line interface, named—enticingly enough—`ledger`. It supports a few reporting commands, and a large number of options for refining the output from those commands. The basic syntax of any ledger command is:

```
$ ledger [OPTIONS...] COMMAND [ARGS...]
```

After the command word there may appear any number of arguments. For most commands, these arguments are regular expressions that cause the output to relate only to postings matching those regular expressions. For the `xact` command, the arguments have a special meaning, described below.

The regular expressions arguments always match the account name that a posting refers to. To match on the payee of the transaction instead, precede the regular expression with '`payee`' or '`@`'. For example, the following balance command reports account totals for rent, food and movies, but only those whose payee matches Freddie:

```
$ ledger bal rent food movies payee freddie
```

or

```
$ ledger bal rent food movies @freddie
```

There are many, many command options available with the `ledger` program, and it takes a while to master them. However, none of them are required to use the basic reporting commands.

8.2 Command Line Quick Reference

8.2.1 Basic Reporting Commands

`balance`
`bal` Show account balances.

`register`
`reg` Show all transactions with running total.

`csv` Show transactions in csv format, for exporting to other programs.

`print` Print transactions in a format readable by ledger.

`xml` Produce XML output of the register command.

`lisp`
`emacs` Produce s-expression output, suitable for Emacs.

`equity` Print account balances as transactions.

`prices` Print price history for matching commodities.

`pricedb` Print price history for matching commodities in a format readable by ledger.

`xact` Generate transactions based on previous postings.

8.2.2 Basic Options

`--help`
`-h` Print summary of all options.

`--version`
`-v` Print version information and exit.

`--file` *FILE*
`-f` *FILE* Read `FILE` as a ledger file.

`--output` *FILE*
`-o` *FILE* Redirect output to `FILE`.

`--init-file` *FILE*
`-i` *FILE* Specify an options file.

`--account` *STR*
`-a` *STR* Specify default account *STR* for QIF file postings.

8.2.3 Report Filtering

`--current`
`-c` Display only transactions on or before the current date.

`--begin` *DATE*
`-b` *DATE* Limit the processing to transactions on or after *DATE*.

`--end` *DATE*
`-e` *DATE* Limit the processing to transactions before *DATE*.

`--period` *PERIOD_EXPRESSION*
`-p` *PERIOD_EXPRESSION*
Limit the processing to transactions in *PERIOD_EXPRESSION*.

`--period-sort` *VEXPR*
Sort postings within each period according to *VEXPR*.

`--cleared`
`-C` Display only cleared postings.

`--dc` Display register or balance in debit/credit format.

`--uncleared`
`-U` Display only uncleared postings.

`--real`
`-R` Display only real postings.

`--actual`
`-L` Display only actual postings, not automated ones.

`--related`
`-r` Display related postings.

`--budget` Display how close your postings meet your budget.

`--add-budget`
: Show unbudgeted postings.

`--unbudgeted`
: Show only unbudgeted postings.

`--forecast` *VEXPR*
: Project balances into the future.

`--limit` *EXPR*
`-l` *EXPR* : Limit which postings are used in calculations by *EXPR*.

`--amount` *EXPR*
`-t` *EXPR* : Change value expression reported in `register` report.

`--total` *VEXPR*
`-T` *VEXPR* : Change the value expression used for "totals" column in `register` and `balance` reports.

8.2.4 Error Checking and Calculation Options

`--strict` : Accounts, tags or commodities not previously declared will cause warnings.

`--pedantic`
: Accounts, tags or commodities not previously declared will cause errors.

`--check-payees`
: Enable strict and pedantic checking for payees as well as accounts, commodities and tags. This only works in conjunction with `--strict` or `--pedantic`.

`--immediate`
: Instruct ledger to evaluate calculations immediately rather than lazily.

8.2.5 Output Customization

`--collapse`
`-n` : Collapse transactions with multiple postings.

`--subtotal`
`-s` : Report register as a single subtotal.

`--by-payee`
`-P` : Report subtotals by payee.

`--empty`
`-E` : Include empty accounts in the report.

`--weekly`
`-W` : Report posting totals by week.

`--quarterly`
: Report posting totals by quarter.

`--yearly`
`-Y` : Report posting totals by year.

`--dow` : Report posting totals by day of week.

`--sort` *VEXPR*
`-S` *VEXPR* Sort a report using *VEXPR*.

`--wide`
`-w` Assume 132 columns instead of 80.

`--head` *INT*
Report the first *INT* postings.

`--tail` *INT*
Report the last *INT* postings.

`--pager` *FILE*
Direct output to *FILE* pager program.

`--average`
`-A` Report the average posting value.

`--deviation`
`-D` Report each posting's deviation from the average.

`--percent`
`-%` Show subtotals in the balance report as percentages.

`--pivot` *TAG*
Produce a pivot table of the *TAG* type specified.

`--amount-data`
`-j` Show only the date and value columns to format the output for plots.

`--plot-amount-format` *FORMAT_STRING*
Specify the format for the plot output.

`--total-data`
`-J` Show only the date and total columns to format the output for plots.

`--plot-total-format` *FORMAT_STRING*
Specify the format for the plot output.

`--display` *EXPR*
`-d` *EXPR* Display only postings that meet the criteria in the *EXPR*.

`--date-format` *DATE_FORMAT*
`-y` *DATE_FORMAT*
Change the basic date format used in reports.

`--format` *FORMAT_STRING*
`--balance-format` *FORMAT_STRING*
`--register-format` *FORMAT_STRING*
`--prices-format` *FORMAT_STRING*
`-F` *FORMAT_STRING*
Set the reporting format for various reports.

`--anon` Print the ledger register with anonymized accounts and payees, useful for filing bug reports.

8.2.6 Grouping Options

`--by-payee`
`-P` Group postings by common payee names.

`--daily`
`-D` Group postings by day.

`--weekly`
`-W` Group postings by week.

`--monthly`
`-M` Group postings by month.

`--quarterly`
Group postings by quarter.

`--yearly`
`-Y` Group postings by year.

`--dow` Group by day of weeks.

`--subtotal`
`-s` Group postings together, similar to the balance report.

8.2.7 Commodity Reporting

`--price-db` *FILE*
Use `FILE` for retrieving stored commodity prices.

`--price-exp` *INT*
`-Z` *INT* Set expected freshness of prices in *INT* minutes.

`--download`
`-Q` Download quotes using the script named `getquote`.

`--getquote` *FILE*
Sets the path to a user-defined script to download commodity prices.

`--quantity`
`-O` Report commodity totals without conversion.

`--basis`
`-B` Report cost basis.

`--market`
`-V` Report last known market value.

`--gain`
`-G` Report net gain or loss for commodities that have a price history.

8.3 Detailed Option Description

8.3.1 Global Options

Options for Ledger reports affect three separate scopes of operation: Global, Session, and Report. In practice there is very little difference between these scopes. Ledger 3.0 contains provisions for GUIs, which would make use of the different scopes by keeping an instance of Ledger running in the background and running multiple sessions with multiple reports per session.

`--args-only`
: Ignore all environment and init-file settings and use only command-line arguments to control Ledger. Useful for debugging or testing small journal files not associated with your main financial database.

`--debug` *`CODE`*
: FIX THIS ENTRY

`--help`
`-h`
: Display the man page for ledger.

`--init-file` *`FILE`*
: Specify the location of the init file. The default is `~/.ledgerrc`.

`--options`
: Display the options in effect for this Ledger invocation, along with their values and the source of those values, for example:

```
$ ledger --options bal --cleared -f ~/ledger/test/input/drewr3.dat
================================================================================
[Global scope options]

[Session scope options]
                  --file = ~/ledger/test/input/drewr3.dat      -f
              --price-db = ~/FinanceData/PriceDB               $price-db

[Report scope options]
                --cleared                                      --cleared
                  --color                                      ?normalize
           --date-format = %Y/%m/%d                            $date-format
                 --limit = cleared                             --cleared
         --prepend-width = 0                                   ?normalize
            --meta-width = 0                                   ?normalize
            --date-width = 10                                  ?normalize
           --payee-width = 21                                  ?normalize
         --account-width = 21                                  ?normalize
          --amount-width = 12                                  ?normalize
           --total-width = 12                                  ?normalize
================================================================================
            $ 775.00  Assets:Checking
         $ -1,000.00  Equity:Opening Balances
            $ 225.00  Expenses:Food:Groceries
--------------------
                   0
```

For the source column, a value starting with a '`-`' or '`--`' indicated the source was a command line argument. If the entry starts with a '`$`', the source was an environment variable. If the source is `?normalize` the value was set internally by ledger, in a function called `normalize_options`.

`--script` *FILE*
: Execute a ledger script.

`--trace` *INT*
: FIX THIS ENTRY

`--verbose`
`-v`
: FIX THIS ENTRY

`--verify`
: FIX THIS ENTRY

`--verify-memory`
: FIX THIS ENTRY

`--version`
: FIX THIS ENTRY

8.3.2 Session Options

Options for Ledger reports affect three separate scopes of operation: Global, Session, and Report. In practice there is very little difference between these scopes. Ledger 3.0 contains provisions for GUIs, which would make use of the different scopes by keeping an instance of Ledger running in the background and running multiple sessions with multiple reports per session.

`--cache` *FIXME*
: FIX THIS ENTRY

`--check-payees`
: FIX THIS ENTRY

`--day-break`
: FIX THIS ENTRY

`--decimal-comma`
: Direct Ledger to parse journals using the European standard comma as a decimal separator, not the usual period.

`--download`
`-Q`
: Direct Ledger to download prices using the script defined via the option `--getquote` *FILE*.

`--explicit`
: FIX THIS ENTRY

`--file` *FILE*
`-f` *FILE*
: Specify the input `FILE` for this invocation.

`--getquote` *FILE*
: Tell ledger where to find the user defined script to download prices information.

`--input-date-format` *DATE_FORMAT*
: Specify the input date format for journal entries. For example,

```
$ ledger convert Export.csv --input-date-format "%m/%d/%Y"
```

Would convert the `Export.csv` file to ledger format, assuming the dates in the CSV file are like 12/23/2009 (see Section 12.5.5 [Date and Time Format Codes], page 104).

`--master-account` *STR*

Prepend all account names with the argument.

```
$ ledger -f drewr3.dat bal --no-total --master-account HUMBUG
                   0  HUMBUG
          $ -3,804.00    Assets
           $ 1,396.00      Checking
              $ 30.00        Business
          $ -5,200.00      Savings
          $ -1,000.00    Equity:Opening Balances
           $ 6,654.00    Expenses
           $ 5,500.00      Auto
              $ 20.00      Books
             $ 300.00      Escrow
             $ 334.00      Food:Groceries
             $ 500.00      Interest:Mortgage
          $ -2,030.00    Income
          $ -2,000.00      Salary
             $ -30.00      Sales
             $ 180.00    Liabilities
             $ -20.00      MasterCard
             $ 200.00      Mortgage:Principal
```

`--no-aliases`

Ledger does not expand any aliases if this option is specified.

`--pedantic`

Accounts, tags or commodities not previously declared will cause errors.

`--permissive`

FIX THIS ENTRY

`--price-db` *FILE*

Specify the location of the price entry data file.

`--price-exp` *INT*

`-Z` *INT*

`--leeway` *INT*

Set the expected freshness of price quotes, in *INT* minutes. That is, if the last known quote for any commodity is older than this value, and if `--download` is being used, then the Internet will be consulted again for a newer price. Otherwise, the old price is still considered to be fresh enough.

`--strict` Ledger normally silently accepts any account or commodity in a posting, even if you have misspelled a commonly used one. The option `--strict` changes that behavior. While running with `--strict`, Ledger interprets all cleared transactions as correct, and if it encounters a new account or commodity (same as a misspelled commodity or account) it will issue a warning giving you the file and line number of the problem.

`--recursive-aliases`

Normally, ledger only expands aliases once. With this option, ledger tries to expand the result of alias expansion recursively, until no more expansions apply.

`--time-colon`

The `--time-colon` option will display the value for a seconds based commodity as real hours and minutes.

For example 8100 seconds by default will be displayed as 2.25 whereas with the `--time-colon` option they will be displayed as 2:15.

`--value-expr` *FIXME*
: FIX THIS ENTRY

8.3.3 Report Options

Options for Ledger reports affect three separate scopes of operation: Global, Session, and Report. In practice there is very little difference between these scopes. Ledger 3.0 contains provisions for GUIs, which would make use of the different scopes by keeping an instance of Ledger running in the background and running multiple sessions with multiple reports per session.

`--abbrev-len` *INT*
: Set the minimum length an account can be abbreviated to if it doesn't fit inside the `account-width`. If *INT* is zero, then the account name will be truncated on the right. If *INT* is greater than `account-width` then the account will be truncated on the left, with no shortening of the account names in order to fit into the desired width.

`--account` *STR*
: Prepend *STR* to all accounts reported. That is, the option '`--account Personal`' would tack '`Personal:`' to the beginning of every account reported in a balance report or register report.

`--account-width` *INT*
: Set the width of the account column in the `register` report to *INT* characters.

`--actual`
`-L`
: Report only real transactions, ignoring all automated or virtual transactions.

`--add-budget`
: Show only unbudgeted postings.

`--amount` *EXPR*
`-t` *EXPR*
: Apply the given value expression to the posting amount (see Chapter 11 [Value Expressions], page 96). Using `--amount` *EXPR* you can apply an arbitrary transformation to the postings.

`--amount-data`
`-j`
: On a register report print only the date and amount of postings. Useful for graphing and spreadsheet applications.

`--amount-width` *INT*
: Set the width in characters of the amount column in the `register` report.

`--anon`
: Anonymize registry output, mostly for sending in bug reports.

`--auto-match`
: FIX THIS ENTRY

`--aux-date`
`--effective`
: Show auxiliary dates for all calculations (see Section 5.22.7 [Effective Dates], page 44).

`--average`
`-A` Print average values over the number of transactions instead of running totals.

`--balance-format` *FORMAT_STRING*
Specify the format to use for the `balance` report (see Chapter 12 [Format Strings], page 101). The default is:

```
"%(justify(scrub(display_total), 20, -1, true, color))"
"  %(!options.flat ? depth_spacer : \"\")"
"%-(ansify_if(partial_account(options.flat), blue if color))\n%/"
"%$1\n%/"
"--------------------\n"
```

`--base` FIX THIS ENTRY

`--basis`
`-B`
`--cost` Report the cost basis on all posting.

`--begin` *DATE*
Specify the start *DATE* of all calculations. Transactions before that date will be ignored.

`--bold-if` *VEXPR*
Print the entire line in bold if the given value expression is true (see Chapter 11 [Value Expressions], page 96).

```
$ ledger reg Expenses --begin Dec --bold-if "amount>100"
```

list all transactions since the beginning of December and print in bold any posting greater than $100.

`--budget` Only display budgeted items. In a register report this displays transactions in the budget, in a balance report this displays accounts in the budget (see Chapter 9 [Budgeting and Forecasting], page 93).

`--budget-format` *FORMAT_STRING*
Specify the format to use for the `budget` report (see Chapter 12 [Format Strings], page 101). The default is:

```
"%(justify(scrub(display_total), 20, -1, true, color))"
"  %(!options.flat ? depth_spacer : \"\")"
"%-(ansify_if(partial_account(options.flat), blue if color))\n%/"
"%$1\n%/"
"--------------------\n"
```

`--by-payee`
`-P` Group the register report by payee.

`--cleared`
`-C` Consider only transactions that have been cleared for display and calculation.

`--cleared-format` *FORMAT_STRING*
FIX THIS ENTRY Specify the format to use for the `cleared` report (see Chapter 12 [Format Strings], page 101). The default is:

```
"%(justify(scrub(get_at(total_expr, 0)), 16, 16 + prepend_width, "
" true, color))  %(justify(scrub(get_at(total_expr, 1)), 18, "
" 36 + prepend_width, true, color))"
```

```
"    %(latest_cleared ? format_date(latest_cleared) : \"         \")"
"    %(!options.flat ? depth_spacer : \"\")"
"%-(ansify_if(partial_account(options.flat), blue if color))\n%/"
"%$1  %$2    %$3\n%/"
"%(prepend_width ? \" \" * prepend_width : \"\")"
"----------------    ----------------    ---------\n"
```

`--collapse`
`-n` By default ledger prints all accounts in an account tree. With `--collapse` it prints only the top level account specified.

`--collapse-if-zero`
Collapse the account display only if it has a zero balance.

`--color`
`--ansi` Use color if the terminal supports it.

`--columns` *INT*
Specify the width of the `register` report in characters.

`--count` Direct ledger to report the number of items when appended to the `commodities`, `accounts` or `payees` command.

`--csv-format` *FORMAT_STRING*
Specify the format to use for the `csv` report (see Chapter 12 [Format Strings], page 101). The default is:

```
"%(quoted(date)),"
"%(quoted(code)),"
"%(quoted(payee)),"
"%(quoted(display_account)),"
"%(quoted(commodity(scrub(display_amount)))),"
"%(quoted(quantity(scrub(display_amount)))),"
"%(quoted(cleared ? \"*\" : (pending ? \"!\" : \"\"))),"
"%(quoted(join(note | xact.note)))\n"
```

`--current`
Shorthand for '`--limit "date <= today"`'.

`--daily`
`-D` Shorthand for '`--period "daily"`'.

`--date` *EXPR*
Transform the date of the transaction using *EXPR*.

`--date-format` *DATE_FORMAT*
`-y` *DATE_FORMAT*
Specify the format ledger should use to read and print dates (see Section 12.5.5 [Date and Time Format Codes], page 104).

`--date-width` *INT*
Specify the width, in characters, of the date column in the `register` report.

`--datetime-format` *FIXME*
FIX THIS ENTRY

`--dc` Display register or balance in debit/credit format If you use `--dc` with either the `register` (reg) or `balance` (bal) commands, you will now get extra columns. The register goes from this:

```
12-Mar-10 Employer           Assets:Cash               $100        $100
                             Income:Employer          $-100           0
12-Mar-10 KFC                Expenses:Food             $20          $20
                             Assets:Cash              $-20           0
12-Mar-10 KFC - Rebate       Assets:Cash               $5           $5
                             Expenses:Food             $-5           0
12-Mar-10 KFC - Food & Reb.. Expenses:Food             $20          $20
                             Expenses:Food             $-5          $15
                             Assets:Cash              $-15           0
```

To this:

```
12-Mar-10 Employer       Assets:Cash        $100          0        $100
                         In:Employer          0         $100         0
12-Mar-10 KFC            Expens:Food         $20          0        $20
                         Assets:Cash          0          $20         0
12-Mar-10 KFC - Rebate   Assets:Cash          $5          0         $5
                         Expens:Food          0           $5         0
12-Mar-10 KFC - Food &.. Expens:Food         $20          0        $20
                         Expens:Food          0           $5        $15
                         Assets:Cash          0          $15         0
```

Where the first column is debits, the second is credits, and the third is the running total. Only the running total may contain negative values.

For the balance report without `--dc`:

```
                  $70  Assets:Cash
                  $30  Expenses:Food
                $-100  Income:Employer
--------------------
                   0
```

And with `--dc` it becomes this:

```
            $105            $35            $70  Assets:Cash
             $40            $10            $30  Expenses:Food
               0           $100          $-100  Income:Employer
-----------------------------------------------
            $145           $145              0
```

`--depth` *INT*

Limit the depth of the account tree. In a balance report, for example, a '`--depth 2`' statement will print balances only for accounts with two levels, i.e. '`Expenses:Entertainment`' but not '`Expenses:Entertainment:Dining`'. This is a display predicate, which means it only affects display, not the total calculations.

`--deviation`

Report each posting's deviation from the average. It is only meaningful in the register and prices reports.

`--display` *EXPR*

Display only lines that satisfy the expression *EXPR*.

`--display-amount` *EXPR*

Apply a transformation to the *displayed* amount. This happens after calculations occur.

`--display-total` *EXPR*
: Apply a transformation to the *displayed* total. This happens after calculations occur.

`--dow`
`--days-of-week`
: Group transactions by the day of the week.

```
$ ledger reg Expenses --dow --collapse
```

Will print all Expenses totaled for each day of the week.

`--empty`
`-E`
: Include empty accounts in the report and in average calculations.

`--end` *DATE*
: Specify the end *DATE* for a transaction to be considered in the report. All transactions on or after this date are ignored.

`--equity`
: Related to the `equity` command (see Section 7.1.2 [The `equity` command], page 53). Gives current account balances in the form of a register report.

`--exact`
: FIX THIS ENTRY

`--exchange` *COMMODITY*
`-X` *COMMODITY*
: Display values in terms of the given *COMMODITY*. The latest available price is used. The syntax `-X` *COMMODITY1*`:`*COMMODITY2* displays values in *COMMODITY1* in terms of *COMMODITY2* using the latest available price, but will not automatically covert any other commodities to *COMMODITY2*. Multiple `-X` arguments may be used on a single command line (as in `-X COMMODITY1:COMMODITY2 -X COMMODITY3:COMMODITY2`), which is particularly useful for situations where many prices are available for reporting in terms of *COMMODITY2*, but only a few should be displayed that way.

`--flat`
: Force the full names of accounts to be used in the balance report. The balance report will not use an indented tree.

`--force-color`
: Output TTY color codes even if the TTY doesn't support them. Useful for TTYs that don't advertise their capabilities correctly.

`--force-pager`
: Force Ledger to paginate its output.

`--forecast-while` *VEXPR*
`--forecast` *VEXPR*
: Continue forecasting while *VEXPR* is true.

`--forecast-years` *INT*
: Forecast at most *INT* years into the future.

`--format` *FORMAT_STRING*
`-F` *FORMAT_STRING*
: Use the given format string to print output.

`--gain`
`-G`
`--change` Report on gains using the latest available prices.

`--generated`
Include auto-generated postings (such as those from automated transactions) in the report, in cases where you normally wouldn't want them.

`--group-by` *EXPR*
Group transactions together in the `register` report. *EXPR* can be anything, although most common would be `payee` or `commodity`. The `tags()` function is also useful here.

`--group-title-format` *FORMAT_STRING*
Set the format for the headers that separates the report sections of a grouped report. Only has an effect with a `--group-by` *EXPR* register report.

```
$ ledger reg Expenses --group-by "payee" --group-title-format "-----------
------------ %-20(value) --------------------\n"
------------------------ 7-Eleven             ---------------------
2011/08/13 7-Eleven              Expenses:Auto:Misc          $ 5.80       $ 5.80

------------------------ AAA Dues             ---------------------
2011/06/02 AAA Dues              Expenses:Auto:Misc        $ 215.00     $ 215.00

------------------------ ABC Towing and Wrecking ----------------------
2011/03/17 ABC Towing and Wrec.. Expenses:Auto:Hobbies      $ 48.20      $ 48.20
...
```

`--head` *INT*
`--first` *INT*
Print the first *INT* entries. Opposite of `--tail` *INT*.

`--historical`
`-H` FIX THIS ENTRY

`--immediate`
FIX THIS ENTRY

`--inject` Use `Expected` amounts in calculations. In case you know what amount a transaction should be, but the actual transaction has the wrong value you can use metadata to specify the expected amount:

```
2012-03-12  Paycheck
    Income  $-990;  Expected:: $-1000.00
    Checking
```

Then using the command `ledger reg --inject=Expected Income` would treat the transaction as if the "Expected Value" was actual.

`--invert` Change the sign of all reported values.

`--limit` *EXPR*
`-l` *EXPR* Only transactions that satisfy *EXPR* are considered in calculations and for display.

`--lot-dates`
Report the date on which each commodity in a balance report was purchased.

`--lot-notes`
`--lot-tags`
: Report the tag attached to each commodity in a balance report.

`--lot-prices`
: Report the price at which each commodity in a balance report was purchased.

`--lots`
: Report the date and price at which each commodity was purchased in a balance report.

`--lots-actual`
: FIX THIS ENTRY

`--market`
`-V`
: Use the latest market value for all commodities.

`--meta` *TAG*
: In the register report, prepend the transaction with the value of the given *TAG*.

`--meta-width` *INT*
: Specify the width of the Meta column used for the `--meta` *TAG* options.

`--monthly`
`-M`
: Synonym for '`--period "monthly"`'.

`--no-aliases`
: Aliases are completely ignored.

`--no-color`
: Suppress any color TTY output.

`--no-rounding`
: Don't output '`<Rounding>`' postings. Note that this will cause the running total to often not add up! Its main use is for `--amount-data (-j)` and `--total-data (-J)` reports.

`--no-titles`
: Suppress the output of group titles.

`--no-total`
: Suppress printing the final total line in a balance report.

`--now` *DATE*
: Define the current date in case you want to calculate in the past or future using `--current`.

`--only` *FIXME*
: This is a postings predicate that applies after certain transforms have been executed, such as periodic gathering.

`--output` *FILE*
: Redirect the output of ledger to the file defined in `FILE`.

`--pager` *FILE*
: Specify the pager program to use.

`--payee` *VEXPR*
: Sets a value expression for formatting the payee. In the `register` report this prevents the second entry from having a date and payee for each transaction.

`--payee-width` *INT*
: Set the number of columns dedicated to the payee in the register report to *INT*.

`--pending`
: Use only postings that are marked pending.

`--percent`
`-%`
: Calculate the percentage value of each account in balance reports. Only works for accounts that have a single commodity.

`--period` *PERIOD_EXPRESSION*
: Define a period expression that sets the time period during which transactions are to be accounted. For a `register` report only the transactions that satisfy the period expression with be displayed. For a `balance` report only those transactions will be accounted in the final balances.

`--pivot` *TAG*
: Produce a balance pivot report *around* the given *TAG*. For example, if you have multiple cars and track each fuel purchase in '`Expenses:Auto:Fuel`' and tag each fuel purchase with a tag identifying which car the purchase was for '`; Car: Prius`', then the command:

```
$ ledger bal Fuel --pivot "Car"  --period "this year"
            $ 3491.26  Car
            $ 1084.22    M3:Expenses:Auto:Fuel
             $ 149.65    MG V11:Expenses:Auto:Fuel
             $ 621.89    Prius:Expenses:Auto:Fuel
            $ 1635.50    Sienna:Expenses:Auto:Fuel
              $ 42.69  Expenses:Auto:Fuel
--------------------
            $ 3533.95
```

  See Section 5.7.2 [Metadata values], page 35.

`--plot-amount-format` *FORMAT_STRING*
: Define the output format for an amount data plot. See Section 6.4.2 [Visualizing with Gnuplot], page 51.

`--plot-total-format` *FORMAT_STRING*
: Define the output format for a total data plot. See Section 6.4.2 [Visualizing with Gnuplot], page 51.

`--prepend-format` *FORMAT_STRING*
: Prepend *STR* to every line of the output.

`--prepend-width` *INT*
: Reserve *INT* spaces at the beginning of each line of the output.

`--price`
`-I`
: Use the price of the commodity purchase for performing calculations.

`--pricedb-format` *FORMAT_STRING*
: FIX THIS ENTRY

`--prices-format` *FORMAT_STRING*
: FIX THIS ENTRY

`--primary-date`
`--actual-dates`
: Show primary dates for all calculations (see Section 5.22.7 [Effective Dates], page 44).

`--quantity`
`-O`
: Report commodity totals (this is the default).

`--quarterly`
: Synonym for '`--period "quarterly"`'.

`--raw`
: In the `print` report, show transactions using the exact same syntax as specified by the user in their data file. Don't do any massaging or interpreting. This can be useful for minor cleanups, like just aligning amounts.

`--real`
`-R`
: Account using only real transactions ignoring virtual and automatic transactions.

`--register-format` *FORMAT_STRING*
: Define the output format for the `register` report.

`--related`
: In a `register` report show the related account. This is the other *side* of the transaction.

`--related-all`
: Show all postings in a transaction, similar to `--related` but show both *sides* of each transaction.

`--revalued`
: FIX THIS ENTRY

`--revalued-only`
: FIX THIS ENTRY

`--revalued-total` *FIXME*
: FIX THIS ENTRY

`--rich-data`
`--detail` FIX THIS ENTRY

`--seed` *FIXME*
: Set the random seed to *FIXME* for the `generate` command. Used as part of development testing.

`--sort` *VEXPR*
`-S` *VEXPR* Sort the `register` report based on the value expression given to sort.

`--sort-all` *FIXME*
: FIX THIS ENTRY

`--sort-xacts` *VEXPR*
`--period-sort` *VEXPR*
: Sort the postings within transactions using the given value expression.

`--start-of-week` *INT*
: Tell ledger to use a particular day of the week to start its "weekly" summary. '`--start-of-week=1`' specifies Monday as the start of the week.

`--subtotal`
`-s`
: FIX THIS ENTRY

`--tail` *INT*
`--last` *INT*
: Report only the last *INT* entries. Only useful in a `register` report.

`--time-report`
: FIX THIS ENTRY

`--total` *VEXPR*
`-T` *VEXPR*
: Define a value expression used to calculate the total in reports.

`--total-data`
`-J`
: Show only dates and totals to format the output for plots.

`--total-width` *INT*
: Set the width of the total field in the register report.

`--truncate` *CODE*
: Indicates how truncation should happen when the contents of columns exceed their width. Valid arguments are '`leading`', '`middle`', and '`trailing`'. The default is smarter than any of these three, as it considers sub-names within the account name (that style is called "abbreviate").

`--unbudgeted`
: Show only unbudgeted postings.

`--uncleared`
`-U`
: Use only uncleared transactions in calculations and reports.

`--unrealized`
: Show generated unrealized gain and loss accounts in the balance report.

`--unrealized-gains` *STR*
: Allow the user to specify what account name should be used for unrealized gains. Defaults to '`"Equity:Unrealized Gains"`'. Often set in one's `~/.ledgerrc` file to change the default.

`--unrealized-losses` *STR*
: Allow the user to specify what account name should be used for unrealized gains. Defaults to '`"Equity:Unrealized Losses"`'. Often set in one's `~/.ledgerrc` file to change the default.

`--unround`
: Perform all calculations without rounding and display results to full precision.

`--values` Shows the values used by each tag when used in combination with the `tags` command.

`--weekly`
`-W` Synonym for '`--period "weekly"`'.

`--wide` Let the register report use 132 columns instead of 80 (the default). Identical to '`--columns "132"`'.

`--yearly`
`-Y` Synonym for '`--period "yearly"`'.

8.3.4 Basic options

These are the most basic command options. Most likely, the user will want to set them using environment variables (see Section 8.3.8 [Environment variables], page 90), instead of using actual command-line options:

`--help`
`-h` Print a summary of all the options, and what they are used for. This can be a handy way to remember which options do what.

`--version`
Print the current version of ledger and exits. This is useful for sending bug reports, to let the author know which version of ledger you are using.

`--file` *FILE*
`-f` *FILE* Read `FILE` as a ledger file. *FILE* can be '`-`' which is a synonym for '`/dev/stdin`'. This command may be used multiple times. Typically, the environment variable `LEDGER_FILE` is set, rather than using this command-line option.

`--output` *FILE*
`-o` *FILE* Redirect output from any command to `FILE`. By default, all output goes to standard output.

`--init-file` *FILE*
`-i` *FILE* Causes `FILE` to be read by ledger before any other ledger file. This file may not contain any postings, but it may contain option settings. To specify options in the init file, use the same syntax as on the command-line, but put each option on its own line. Here is an example init file:

```
--price-db ~/finance/.pricedb
--wide
; ~/.ledgerrc ends here
```

Option settings on the command-line or in the environment always take precedence over settings in the init file.

`--account` *STR*
`-a` *STR* Specify the default account which QIF file postings are assumed to relate to.

8.3.5 Report filtering

These options change which postings affect the outcome of a report, in ways other than just using regular expressions:

`--current`
`-c` Display only transactions occurring on or before the current date.

`--begin` *DATE*
`-b` *DATE* Constrain the report to transactions on or after *DATE*. Only transactions after that date will be calculated, which means that the running total in the balance report will always start at zero with the first matching transaction. (Note: This is different from using `--display` *EXPR* to constrain what is displayed).

`--end` *DATE*
`-e` *DATE* Constrain the report so that transactions on or after *DATE* are not considered.

`--period` *PERIOD_EXPRESSION*
`-p` *PERIOD_EXPRESSION*
Set the reporting period to *STR*. This will subtotal all matching transactions within each period separately, making it easy to see weekly, monthly, quarterly, etc., posting totals. A period string can even specify the beginning and end of the report range, using simple terms like '`last June`' or '`next month`'. For more details on period expressions, see Section 8.4 [Period Expressions], page 91.

`--period-sort` *VEXPR*
Sort the postings within each reporting period using the value expression *EXPR*. This is most often useful when reporting monthly expenses, in order to view the highest expense categories at the top of each month:

```
$ ledger -M --period-sort total reg ^Expenses
```

`--cleared`
`-C` Display only postings whose transaction has been marked "cleared" (by placing an asterisk to the right of the date).

`--uncleared`
`-U` Display only postings whose transaction has not been marked "cleared" (i.e., if there is no asterisk to the right of the date).

`--real`
`-R` Display only real postings, not virtual. (A virtual posting is indicated by surrounding the account name with parentheses or brackets; see Section 5.8 [Virtual postings], page 35 for more information).

`--actual`
`-L` Display only actual postings, and not those created by automated transactions.

`--related`
`-r` Display postings that are related to whichever postings would otherwise have matched the filtering criteria. In the register report, this shows where money went to, or the account it came from. In the balance report, it shows all the accounts affected by transactions having a related posting. For example, if a file had this transaction:

```
2004/03/20 Safeway
    Expenses:Food                $65.00
    Expenses:Cash                $20.00
    Assets:Checking             $-85.00
```

And the register command was:

```
$ ledger -f example.dat -r register food
```

The following would be printed, showing the postings related to the posting that matched:

```
04-Mar-20 Safeway           Expenses:Cash              $20.00       $20.00
                            Assets:Checking            $-85.00      $-
65.00
```

`--budget` Useful for displaying how close your postings meet your budget. `--add-budget` also shows unbudgeted postings, while `--unbudgeted` shows only those. `--forecast` *VEXPR* is a related option that projects your budget into the future, showing how it will affect future balances. See Chapter 9 [Budgeting and Forecasting], page 93.

`--limit` *EXPR*
`-l` *EXPR* Limit which postings take part in the calculations of a report.

`--amount` *EXPR*
`-t` *EXPR* Change the value expression used to calculate the "value" column in the `register` report, the amount used to calculate account totals in the `balance` report, and the values printed in the `equity` report. See Chapter 11 [Value Expressions], page 96.

`--total` *VEXPR*
`-T` *VEXPR* Set the value expression used for the "totals" column in the `register` and `balance` reports.

8.3.6 Output customization

These options affect only the output, but not which postings are used to create it:

`--collapse`
`-n` Cause transactions in a `register` report with multiple postings to be collapsed into a single, subtotaled transaction.

`--subtotal`
`-s` Cause all transactions in a `register` report to be collapsed into a single, subtotaled transaction.

`--by-payee`
`-P` Report subtotals by payee.

`--empty`
`-E` Include even empty accounts in the `balance` report.

`--weekly`
`-W` Report posting totals by the week. The week begins on whichever day of the week begins the month containing that posting. To set a specific begin date, use a period string, such as '`weekly from DATE`'.

`--monthly`
`-M` Report posting totals by month.

`--yearly`
`-Y` Report posting totals by year. For more complex periods, use `--period`.

`--period` *PERIOD_EXPRESSION*
: Option described above.

`--dow`
: Report posting totals for each day of the week. This is an easy way to see if weekend spending is more than on weekdays.

`--sort` *VEXPR*
`-S` *VEXPR*
: Sort a report by comparing the values determined using the value expression *VEXPR*. For example, using '`-S "-abs(total)"`' in the `balance` report will sort account balances from greatest to least, using the absolute value of the total. For more on how to use value expressions, see Chapter 11 [Value Expressions], page 96.

`--pivot` *TAG*
: Produce a pivot table around the *TAG* provided. This requires meta data using valued tags.

`--wide`
`-w`
: Cause the default `register` report to assume 132 columns instead of 80.

`--head` *INT*
: Cause only the first *INT* transactions to be printed. This is different from using the command-line utility `head`, which would limit to the first *INT* postings. `--tail` *INT* outputs only the last *INT* transactions. Both options may be used simultaneously. If a negative amount is given, it will invert the meaning of the flag (instead of the first five transactions being printed, for example, it would print all but the first five).

`--pager` *FILE*
: Tell Ledger to pass its output to the given pager program; very useful when the output is especially long. This behavior can be made the default by setting the `LEDGER_PAGER` environment variable.

`--average`
`-A`
: Report the average posting value.

`--deviation`
`-D`
: Report each posting's deviation from the average. It is only meaningful in the `register` and `prices` reports.

`--percent`
`-%`
: Show account subtotals in the `balance` report as percentages of the parent account.

`--amount-data`
`-j`
: Change the `register` report so that it prints nothing but the date and the value column, and the latter without commodities. This is only meaningful if the report uses a single commodity. This data can then be fed to other programs, which could plot the date, analyze it, etc.

`--total-data`
`-J`
: Change the `register` report so that it prints nothing but the date and total columns, without commodities.

`--display` *EXPR*
`-d` *EXPR* Limit which postings or accounts are actually displayed in a report. They might still be calculated, and be part of the running total of a register report, for example, but they will not be displayed. This is useful for seeing last month's checking postings, against a running balance which includes all posting values:

```
$ ledger -d "d>=[last month]" reg checking
```

The output from this command is very different from the following, whose running total includes only postings from the last month onward:

```
$ ledger -p "last month" reg checking
```

Which is more useful depends on what you're looking to know: the total amount for the reporting range (using `--period` *PERIOD_EXPRESSION* `(-p)`), or simply a display restricted to the reporting range (using `--display` *EXPR* `(-d)`).

`--date-format` *DATE_FORMAT*
`-y` *DATE_FORMAT*
Change the basic date format used by reports. The default uses a date like '`2004/08/01`', which represents the default date format of `%Y/%m/%d`. To change the way dates are printed in general, the easiest way is to put `--date-format` *DATE_FORMAT* in the Ledger initialization file `~/.ledgerrc` (or the file referred to by `LEDGER_INIT`).

`--format` *FORMAT_STRING*
`-F` *FORMAT_STRING*
Set the reporting format for whatever report ledger is about to make. See Chapter 12 [Format Strings], page 101. There are also specific format commands for each report type:

`--balance-format` *FORMAT_STRING*
Define the output format for the `balance` report. The default (defined in `report.h` is:

```
"%(ansify_if(
        justify(scrub(display_total), 20,
                20 + int(prepend_width), true, color),
                  bold if should_bold))
          %(!options.flat ? depth_spacer : \"\")
       %-(ansify_if(
          ansify_if(partial_account(options.flat), blue if color),
                    bold if should_bold))\n%/
       %$1\n%/
       %(prepend_width ? \" \" * int(prepend_width) : \"\")
       --------------------\n"
```

`--cleared-format` *FORMAT_STRING*
Define the format for the cleared report. The default is:

```
"%(justify(scrub(get_at(display_total, 0)), 16, 16 + int(prepend_width),
       true, color))  %(justify(scrub(get_at(display_total, 1)), 18,
       36 + int(prepend_width), true, color))
          %(latest_cleared ? format_date(latest_cleared) : \"         \")
          %(!options.flat ? depth_spacer : \"\")
       %-(ansify_if(partial_account(options.flat), blue if color))\n%/
       %$1  %$2    %$3\n%/
       %(prepend_width ? \" \" * int(prepend_width) : \"\")
       ---------------  ----------------    ---------\n"
```

`--register-format` *FORMAT_STRING*

Define the output format for the `register` report. The default (defined in `report.h` is:

```
"%(ansify_if(
       ansify_if(justify(format_date(date), int(date_width)),
                 green if color and date > today),
                 bold if should_bold))
      %(ansify_if(
        ansify_if(justify(truncated(payee, int(payee_width)), int(payee_width)),
                  bold if color and !cleared and actual),
                  bold if should_bold))
      %(ansify_if(
        ansify_if(justify(truncated(display_account, int(account_width),
                                    int(abbrev_len)), int(account_width)),
                  blue if color),
                  bold if should_bold))
      %(ansify_if(
        justify(scrub(display_amount), int(amount_width),
                3 + int(meta_width) + int(date_width) + int(payee_width)
                  + int(account_width) + int(amount_width) + int(prepend_width),
                true, color),
                bold if should_bold))
      %(ansify_if(
        justify(scrub(display_total), int(total_width),
                4 + int(meta_width) + int(date_width) + int(payee_width)
                  + int(account_width) + int(amount_width) + int(total_width)
                  + int(prepend_width), true, color),
                bold if should_bold))\n%/
     %(justify(\" \", int(date_width)))
      %(ansify_if(
        justify(truncated(has_tag(\"Payee\") ? payee : \" \",
                          int(payee_width)), int(payee_width)),
                  bold if should_bold))
      %$3 %$4 %$5\n"
```

`--csv-format` *FORMAT_STRING*

Set the format for `csv` reports. The default is:

```
"%(quoted(date)),
 %(quoted(code)),
 %(quoted(payee)),
 %(quoted(display_account)),
 %(quoted(commodity(scrub(display_amount)))),
 %(quoted(quantity(scrub(display_amount)))),
 %(quoted(cleared ? \"*\" : (pending ? \"!\" : \"\"))),
 %(quoted(join(note | xact.note)))\n"
```

`--plot-amount-format` *FORMAT_STRING*

Set the format for amount plots, using the `--amount-data (-j)` option. The default is:

```
"%(format_date(date, \"%Y-%m-%d\")) %(quantity(scrub(display_amount)))\n"
```

`--plot-total-format` *FORMAT_STRING*

Set the format for total plots, using the `--total-data (-J)` option. The default is:

```
"%(format_date(date, \"%Y-%m-%d\")) %(quantity(scrub(display_total)))\n"
```

`--pricedb-format` *FORMAT_STRING*
: Set the format expected for the historical price file. The default is:

```
"P %(datetime) %(display_account) %(scrub(display_amount))\n"
```

`--prices-format` *FORMAT_STRING*
: Set the format for the `prices` report. The default is:

```
"%(date) %-8(display_account) %(justify(scrub(display_amount), 12,
          2 + 9 + 8 + 12, true, color))\n"
```

8.3.7 Commodity reporting

These options affect how commodity values are displayed:

`--price-db` *FILE*
: Set the file that is used for recording downloaded commodity prices. It is always read on startup, to determine historical prices. Other settings can be placed in this file manually, to prevent downloading quotes for a specific commodity, for example. This is done by adding a line like the following:

```
; Don't download quotes for the dollar, or timelog values
N $
N h
```

Note: Ledger NEVER writes output to files. You are responsible for updating the price-db file. The best way is to have your price download script maintain this file.

The format of the file can be changed by telling ledger to use the `--pricedb-format` *FORMAT_STRING* you define.

`--price-exp` *INT*
`-Z` *INT*
: Set the expected freshness of price quotes, in *INT* minutes. That is, if the last known quote for any commodity is older than this value, and if `--download` is being used, then the Internet will be consulted again for a newer price. Otherwise, the old price is still considered to be fresh enough.

`--download`
`-Q`
: Cause quotes to be automagically downloaded, as needed, by running a script named `getquote` and expecting that script to return a value understood by ledger. A sample implementation of a `getquote` script, implemented in Perl, is provided in the distribution. Downloaded quote price are then appended to the price database, usually specified using the environment variable `LEDGER_PRICE_DB`.

There are several different ways that ledger can report the totals it displays. The most flexible way to adjust them is by using value expressions, and the `--amount` *EXPR* `(-t)` and `--total` *VEXPR* `(-T)` options. However, there are also several "default" reports, which will satisfy most users' basic reporting needs:

`--quantity`
`-O`
: Report commodity totals (this is the default).

`--basis`
`-B`
: Report the cost basis for all postings.

`--market`
`-V` Use the last known value for commodities to calculate final values.

`--gain`
`-G` Report the net gain/loss for all commodities in the report that have a price history.

Often you will be more interested in the value of your entire holdings, in your preferred currency. It might be nice to know you hold 10,000 shares of PENNY, but you are more interested in whether or not that is worth $1000.00 or $10,000.00. However, the current day value of a commodity can mean different things to different people, depending on the accounts involved, the commodities, the nature of the transactions, etc.

When you specify `--market (-V)`, or `--exchange` *`COMMODITY`* `(-X)`, you are requesting that some or all of the commodities be valuated as of today (or whatever `--now` *`DATE`* is set to). But what does such a valuation mean? This meaning is governed by the presence of a *VALUE* meta-data property, whose content is an expression used to compute that value.

If no *VALUE* property is specified, each posting is assumed to have a default, as if you'd specified a global, automated transaction as follows:

```
= expr true
    ; VALUE:: market(amount, date, exchange)
```

This definition emulates the present day behavior of `--market (-V)` and `--exchange` *`COMMODITY`* `(-X)` (in the case of '`-X`', the requested commodity is passed via the string '`exchange`' above).

One thing many people have wanted to do is to fixate the valuation of old European currencies in terms of the Euro after a certain date:

```
= expr commodity == "DM"
    ; VALUE:: date < [Jun 2008] ? market(amount, date, exchange) : 1.44 EUR
```

This says: If `--now` *`DATE`* is some old date, use market prices as they were at that time; but if `--now` *`DATE`* is past June 2008, use a fixed price for converting Deutsch Mark to Euro.

Or how about never re-valuating commodities used in Expenses, since they cannot have a different future value:

```
= /^Expenses:/
    ; VALUE:: market(amount, post.date, exchange)
```

This says the future valuation is the same as the valuation at the time of posting. post.date equals the posting's date, while just 'date' is the value of `--now` *`DATE`* (defaults to today).

Or how about valuating miles based on a reimbursement rate during a specific time period:

```
= expr commodity == "miles" and date >= [2007] and date < [2008]
    ; VALUE:: market($1.05, date, exchange)
```

In this case, miles driven in 2007 will always be valuated at $1.05 each. If you use '`-X EUR`' to expressly request all amounts in Euro, Ledger shall convert $1.05 to Euro by whatever means are appropriate for dollars.

Note that you can have a valuation expression specific to a particular posting or transaction, by overriding these general defaults using specific meta-data:

```
2010-12-26 Example
    Expenses:Food                $20
    ; Just to be silly, always valuate *these* $20 as 30 DM, no matter what
    ; the user asks for with -V or -X
    ; VALUE:: 30 DM
    Assets:Cash
```

This example demonstrates that your value expression should be as symbolic as possible, using terms like 'amount' and 'date', rather than specific amounts and dates. Also, you should pass the amount along to the function 'market' so it can be further revalued if the user has asked for a specific currency.

Or, if it better suits your accounting, you can be less symbolic, which allows you to report most everything in EUR if you use '`-X EUR`', except for certain accounts or postings which should always be valuated in another currency. For example:

```
= /^Assets:Brokerage:CAD$/
    ; Always report the value of commodities in this account in
    ; terms of present day dollars, despite what was asked for
    ; on the command-line VALUE:: market(amount, date, '$')
```

Ledger presently has no way of handling such things as FIFO and LIFO.

If you specify an unadorned commodity name, like AAPL, it will balance against itself. If `--lots` are not being displayed, then it will appear to balance against any lot of AAPL.

If you specify an adorned commodity, like AAPL {$10.00}, it will also balance against itself, and against any AAPL if `--lots` is not specified. But if you do specify `--lot-prices`, for example, then it will balance against that specific price for AAPL.

Normally when you use `--exchange` *COMMODITY* `(-X)` to request that amounts be reported in a specific commodity, Ledger uses these values:

- Register Report For the `register` report, use the value of that commodity on the date of the posting being reported, with a '`<Revalued>`' posting added at the end if today's value is different from the value of the last posting.
- Balance Report For the `balance` report, use the value of that commodity as of today.

You can now specify `--historical (-H)` to ask that all valuations for any amount be done relative to the date that amount was encountered.

You can also now use `--exchange` *COMMODITY* `(-X)` (and `--historical (-H)`) in conjunction with `--basis (-B)` and `--price (-I)`, to see valuation reports of just your basis costs or lot prices.

Finally, sometimes, you may seek to only report one (or some subset) of the commodities in terms of another commodity. In this situation, you can use the syntax `--exchange` *COMMODITY1*`:`*COMMODITY2* to request that ledger always display *COMMODITY1* in terms of *COMMODITY2*, but you want no other commodities to be automatically displayed in terms of *COMMODITY2* without additional `--exchange` options. For example, if you wanted to report EUR and BTC in terms of USD, but report all other commodities without conversion to USD, you could use: `--exchange EUR:USD --exchange BTC:USD`.

8.3.8 Environment variables

Every option to ledger may be set using an environment variable. If an option has a long name such `--this-option`, setting the environment variable `LEDGER_THIS_OPTION`

will have the same effect as specifying that option on the command-line. Options on the command-line always take precedence over environment variable settings, however.

Note that you may also permanently specify option values by placing option settings in the file `~/.ledgerrc`, for example:

```
--pager /bin/cat
```

8.4 Period Expressions

A period expression indicates a span of time, or a reporting interval, or both. The full syntax is:

```
[INTERVAL] [BEGIN] [END]
```

The optional *INTERVAL* part may be any one of:

```
every day
every week
every month
every quarter
every year
every N days     # N is any integer
every N weeks
every N months
every N quarters
every N years
daily
weekly
biweekly
monthly
bimonthly
quarterly
yearly
```

After the interval, a begin time, end time, both or neither may be specified. As for the begin time, it can be either of:

```
from <SPEC>
since <SPEC>
```

The end time can be either of:

```
to <SPEC>
until <SPEC>
```

Where *SPEC* can be any of:

```
2004
2004/10
2004/10/1
10/1
october
oct
this week  # or day, month, quarter, year
next week
last week
```

The beginning and ending can be given at the same time, if it spans a single period. In that case, just use *SPEC* by itself. In that case, the period '`oct`', for example, will cover all the days in October. The possible forms are:

```
<SPEC>
in <SPEC>
```

Here are a few examples of period expressions:

```
monthly
monthly in 2004
weekly from oct
weekly from last month
from sep to oct
from 10/1 to 10/5
monthly until 2005
from apr
until nov
last oct
weekly last august
```

9 Budgeting and Forecasting

9.1 Budgeting

Keeping a budget allows you to pay closer attention to your income and expenses, by reporting how far your actual financial activity is from your expectations.

To start keeping a budget, put some periodic transactions (see Section 5.22.8 [Periodic Transactions], page 45) at the top of your ledger file. A periodic transaction is almost identical to a regular transaction, except that it begins with a tilde and has a period expression in place of a payee. For example:

```
~ Monthly
    Expenses:Rent               $500.00
    Expenses:Food               $450.00
    Expenses:Auto:Gas           $120.00
    Expenses:Insurance          $150.00
    Expenses:Phone              $125.00
    Expenses:Utilities          $100.00
    Expenses:Movies              $50.00
    Expenses                    $200.00  ; all other expenses
    Assets

~ Yearly
    Expenses:Auto:Repair        $500.00
    Assets
```

These two periodic transactions give the usual monthly expenses, as well as one typical yearly expense. For help on finding out what your average monthly expenses are for any category, use a command like:

```
$ ledger -p "this year" --monthly --average balance ^expenses
```

The reported totals are the current year's average for each account.

Once these periodic transactions are defined, creating a budget report is as easy as adding `--budget` to the command-line. For example, a typical monthly expense report would be:

```
$ ledger --monthly register ^expenses
```

To see the same report balanced against your budget, use:

```
$ ledger --budget --monthly register ^expenses
```

A budget report includes only those accounts that appear in the budget. To see all expenses balanced against the budget, use `--add-budget`. You can even see only the unbudgeted expenses using `--unbudgeted`:

```
$ ledger --unbudgeted --monthly register ^expenses
```

You can also use these flags with the `balance` command.

9.2 Forecasting

Sometimes it's useful to know what your finances will look like in the future, such as determining when an account will reach zero. Ledger makes this easy to do, using the same periodic transactions as are used for budgeting. An example forecast report can be generated with:

```
$ ledger --forecast "T>{\$-500.00}" register ^assets ^liabilities
```

This report continues outputting postings until the running total is greater than $-500.00. A final posting is always shown, to inform you what the total afterwards would be.

Forecasting can also be used with the `balance` report, but by date only, and not against the running total:

```
$ ledger --forecast "d<[2010]" bal ^assets ^liabilities
```

10 Time Keeping

Ledger directly supports "timelog" entries, which have this form:

```
i 2013/03/28 22:13:00 ACCOUNT[  PAYEE]
o 2013/03/29 03:39:00
```

This records a check-in to the given ACCOUNT, and a check-out. You can be checked-in to multiple accounts at a time, if you wish, and they can span multiple days (use `--day-break` to break them up in the report). The number of seconds between check-in and check-out is accumulated as time to that ACCOUNT. If the checkout uses a capital '`O`', the transaction is marked "cleared". You can use an optional PAYEE for whatever meaning you like.

Now, there are a few ways to generate this information. You can use the `timeclock.el` package, which is part of Emacs. Or you can write a simple script in whichever language you prefer to emit similar information. Or you can use Org mode's time-clocking abilities and the '`org2tc`' script developed by John Wiegley.

These timelog entries can appear in a separate file, or directly in your main ledger file. The initial '`i`' and '`o`' characters count as Ledger "directives", and are accepted anywhere that ordinary transactions are valid.

11 Value Expressions

Ledger uses value expressions to make calculations for many different purposes:

1. The values displayed in reports.
2. For predicates (where truth is anything non-zero), to determine which postings are calculated (option `--limit` *`EXPR`* `(-l)`) or displayed (option `--display` *`EXPR`* `(-d)`).
3. For sorting criteria, to yield the sort key.
4. In the matching criteria used by automated postings.

Value expressions support most simple math and logic operators, in addition to a set of functions and variables.

Display predicates are also very handy with register reports, to constrain which transactions are printed. For example, the following command shows only transactions from the beginning of the current month, while still calculating the running balance based on all transactions:

```
$ ledger -d "d>[this month]" register checking
```

The advantage of this command's complexity is that it prints the running total in terms of all transactions in the register. The following, simpler command is similar, but totals only the displayed postings:

```
$ ledger -b "this month" register checking
```

11.1 Variables

Below are the one letter variables available in any value expression. For the `register` and `print` commands, these variables relate to individual postings, and sometimes the account affected by a posting. For the `balance` command, these variables relate to accounts, often with a subtle difference in meaning. The use of each variable for both is specified.

`t` This maps to whatever the user specified with `--amount` *`EXPR`* `(-t)`. In a `register` report, `--amount` *`EXPR`* `(-t)` changes the value column; in a `balance` report, it has no meaning by default. If `--amount` *`EXPR`* `(-t)` was not specified, the current report style's value expression is used.

`T` This maps to whatever the user specified with `--total` *`VEXPR`* `(-T)`. In a register report, `--total` *`VEXPR`* `(-T)` changes the totals column; in a balance report, this is the value given for each account. If `--total` *`VEXPR`* `(-T)` was not specified, the current report style's value expression is used.

`m` This is always the present moment/date.

11.1.1 Posting/account details

`d` A posting's date, as the number of seconds past the epoch. This is always "today" for an account.

`a` The posting's amount; the balance of an account, without considering children.

`b` The cost of a posting; the cost of an account, without its children.

`v` The market value of a posting or an account, without its children.

`g` The net gain (market value minus cost basis), for a posting or an account, without its children. It is the same as '`v-b`'.

`l` The depth ("level") of an account. If an account has one parent, its depth is one.

`n` The index of a posting, or the count of postings affecting an account.

`X` '`1`' if a posting's transaction has been cleared, '`0`' otherwise.

`R` '`1`' if a posting is not virtual, '`0`' otherwise.

`Z` '`1`' if a posting is not automated, '`0`' otherwise.

11.1.2 Calculated totals

`O` The total of all postings seen so far, or the total of an account and all its children.

`N` The total count of postings affecting an account and all its children.

`B` The total cost of all postings seen so far; the total cost of an account and all its children.

`V` The market value of all postings seen so far, or of an account and all its children.

`G` The total net gain (market value minus cost basis), for a series of postings, or an account and its children. It is the same as '`V-B`'.

11.2 Functions

The available one letter functions are:

`-` Negates the argument.

`U` The absolute (unsigned) value of the argument.

`S` Strips the commodity from the argument.

`A` The arithmetic mean of the argument; '`Ax`' is the same as '`x/n`'.

`P` The present market value of the argument. The syntax '`P(x,d)`' is supported, which yields the market value at time '`d`'. If no date is given, then the current moment is used.

11.3 Operators

The binary and ternary operators, in order of precedence, are:

1. `* /`
2. `+ -`
3. `! < > =`
4. `& | ?:`

11.3.1 Unary Operators

`not neg`

11.3.2 Binary Operators

`== < <= > >= and or + - * / QUERY COLON CONS SEQ DEFINE LOOKUP LAMBDA CALL MATCH`

11.4 Complex expressions

More complicated expressions are possible using:

`"amount == COMMODITY AMOUNT"`
: The amount can be any kind of amount supported by ledger, with or without a commodity. Use this for decimal values.

`/REGEX/`
`account =~ /REGEX/`
: A regular expression that matches against an account's full name. If a posting, this will match against the account affected by the posting.

`@/REGEX/`
`expr payee =~ /REGEX/`
: A regular expression that matches against a transaction's payee name.

`%/REGEX/`
`tag(REGEX)`
: A regular expression that matches against a transaction's tags.

`expr date =~ /REGEX/`
: Useful for specifying a date in plain terms. For example, you could say '`expr date =~ /2014/`'.

`expr comment =~ /REGEX/`
: A regular expression that matches against a posting's comment field. This searches only a posting's field, not the transaction's note or comment field. For example, '`ledger reg "expr" "comment =~ /landline/"`' will match:

```
2014/1/29  Phone bill
    Assets:Checking                          $50.00
    Expenses:Phone                          $-50.00  ; landline bill
```

but will not match:

```
2014/1/29  Phone bill  ; landline bill
    ; landline bill
    Assets:Checking                          $50.00
    Expenses:Phone                          $-50.00
```

To match the latter, use '`ledger reg "expr" "note =~ /landline/"`' instead.

`expr note =~ /REGEX/`
: A regular expression that matches against a transaction's note field. This searches all comments in the transaction, including comments on individual postings. Thus, '`ledger reg "expr" "note =~ /landline/"`' will match both all the three examples below:

```
2014/1/29  Phone bill
    Assets:Checking                          $50.00
    Expenses:Phone                          $-50.00  ; landline bill
2014/1/29  Phone bill  ; landline bill
    Assets:Checking                          $50.00
    Expenses:Phone                          $-50.00
```

```
2014/1/29  Phone bill
    ; landline bill
    Assets:Checking                          $50.00
    Expenses:Phone                          $-50.00
```

`(EXPR)` A sub-expression is nested in parenthesis. This can be useful passing more complicated arguments to functions, or for overriding the natural precedence order of operators.

`expr base =~ /REGEX/`
A regular expression that matches against an account's base name. If a posting, this will match against the account affected by the posting.

`expr code =~ /REGEX/`
A regular expression that matches against the transaction code (the text that occurs between parentheses before the payee).

The '`query`' command can be used to see how Ledger interprets your query. This can be useful if you are not getting the results you expect. See see Section 14.3.5 [Pre-Commands], page 118 for more.

11.4.1 Miscellaneous

`abs--> U`

`amount_expr`
`ansify_if`
`ceiling` Return the next integer toward +infinity

`code` Return the transaction code, the string between the parenthesis after the date.

`commodity`
`date`

`display_amount --> t`
`display_total --> T`
`floor` Return the next integer toward -infinity

`format`

`format_date`
`format_datetime`
`get_at`

`is_seq`

`join`

`justify`

`market --> P`
`nail_down`
`now --> d m`
`options`

`percent`

`print`

`quantity`

`quoted`

`round`

`rounded`

`roundto` Return value rounded to n digits. Does not affect formatting.

`scrub`

`should_bold`
`strip --> S`
`to_amount`
`to_balance`
`to_boolean`
`to_date`

`to_datetime`
`to_int`

`to_mask`

`to_sequence`
`to_spring`
`today`

`top_amount`
`total_expr`
`trim`

`truncated`
`unround`

`unrounded`
`value_date`

12 Format Strings

12.1 Format String Basics

Format strings may be used to change the output format of reports. They are specified by passing a formatting string to the `--format` *FORMAT_STRING* `(-F)` option. Within that string, constructs are allowed which make it possible to display the various parts of an account or posting in custom ways.

There are several additional flags that allow you to define formats for specific reports. These are useful to define in your configuration file and will allow you to run ledger reports from the command line without having to enter a new format for each command.

- `--balance-format` *FORMAT_STRING*
- `--budget-format` *FORMAT_STRING*
- `--cleared-format` *FORMAT_STRING*
- `--csv-format` *FORMAT_STRING*
- `--plot-amount-format` *FORMAT_STRING*
- `--plot-total-format` *FORMAT_STRING*
- `--pricedb-format` *FORMAT_STRING*
- `--prices-format` *FORMAT_STRING*
- `--register-format` *FORMAT_STRING*

12.2 Format String Structure

Within a format string, a substitution is specified using a percent '`%`' character. The basic format of all substitutions is:

```
%[-][MIN WIDTH][.MAX WIDTH](VALEXPR)
```

If the optional minus sign '`-`' follows the percent character '`%`', whatever is substituted will be left justified. The default is right justified. If a minimum width is given next, the substituted text will be at least that wide, perhaps wider. If a period and a maximum width is given, the substituted text will never be wider than this, and will be truncated to fit. Here are some examples:

`%-20P` A transaction's payee, left justified and padded to 20 characters wide.

`%20P` The same, right justified, at least 20 chars wide.

`%.20P` The same, no more than 20 chars wide.

The expression following the format constraints can be a single letter, or an expression enclosed in parentheses or brackets.

12.3 Format Expressions

The allowable expressions are:

`%` Inserts a percent sign.

`t` Inserts the results of the value expression specified by `--amount` *`EXPR`* `(-t)`. If `--amount` *`EXPR`* `(-t)` was not specified, the current report style's value expression is used.

`T` Inserts the results of the value expression specified by `--total` *`VEXPR`* `(-T)`. If `--total` *`VEXPR`* `(-T)` was not specified, the current report style's value expression is used.

`(EXPR)` Inserts the amount resulting from the value expression given in parentheses. To insert five times the total value of an account, for example, one could say '`%12(5*O)`'. Note: It's important to put the five first in that expression, so that the commodity doesn't get stripped from the total.

`[DATEFMT]`
Inserts the result of formatting a posting's date with a date format string, exactly like those supported by `strftime`. For example: '`%[%Y/%m/%d %H:%M:%S]`'.

`S` Insert the path name of the file from which the transaction's data was read. Only sensible in a `register` report.

`B` Inserts the beginning character position of that transaction within the file.

`b` Inserts the beginning line of that transaction within the file.

`E` Inserts the ending character position of that transaction within the file.

`e` Inserts the ending line of that transaction within the file.

`D` Returns the date according to the default format.

`d` Returns the date according to the default format. If the transaction has an effective date, it prints `ACTUAL_DATE=EFFECTIVE_DATE`.

`X` If a posting has been cleared, this returns a 1, otherwise returns 0.

`Y` This is the same as '`%X`', except that it only displays a state character if all of the member postings have the same state.

`C` Inserts the transaction code. This is the value specified between parentheses on the first line of the transaction.

`P` Inserts the payee related to a posting.

`A` Inserts the full name of an account.

`N` Inserts the note associated with a posting, if one exists.

`/` The '`%/`' construct is special. It separates a format string between what is printed for the first posting of a transaction, and what is printed for all subsequent postings. If not used, the same format string is used for all postings.

12.4 Balance format

As an example of how flexible the `--format` *`FORMAT_STRING`* strings can be, the default balance format looks like this (the various functions are described later):

```
"%(justify(scrub(display_total), 20, -1, true, color))"
"  %(!options.flat ? depth_spacer : \"\")"
"%-(ansify_if(partial_account(options.flat), blue if color))\n%/"
"%$1\n%/"
"--------------------\n"
```

12.5 Formatting Functions and Codes

12.5.1 Field Widths

The following codes return the width allocated for the specific fields. The defaults can be changed using the corresponding command line options:

- `date_width`
- `payee_width`
- `account_width`
- `amount_width`
- `total_width`

12.5.2 Colors

The character-based formatting ledger can do is limited to the ANSI terminal character colors and font highlights in a normal TTY session.

`red`	`magenta`	`bold`
`green`	`cyan`	`underline`
`yellow`	`white`	`blink`
`blue`	`black`	

12.5.3 Quantities and Calculations

`amount_expr`
`abs`

`commodity`
`display_amount`
`display_total`
`floor`

`get_at`

`is_seq`

`market`

`percent`

`price`

`quantity`

`rounded`

`truncated`
`total_expr`
`top_amount`
`to_boolean`
`to_int`

`to_amount`
`to_balance`
`unrounded`

12.5.4 Date Functions

The following functions allow you to manipulate and format dates.

`date` Return the date of the current transaction.

`format_date(date, "FORMAT_STRING")`
Format the date using the given format string.

`now` Return the current date and time. If the `--now` *`DATE`* option is defined it will return that value.

`today` Return the current date. If the `--now` *`DATE`* option is defined it will return that value.

`to_datetime`
Convert a string to a date-time value.

`to_date` Convert a string to date value.

`value_date`

12.5.5 Date and Time Format Codes

Date and time format are specified as strings of single letter codes preceded by percent signs. Any separator, or no separator can be specified.

12.5.5.1 Days

Dates are formed from a combination of day, month and year codes, in whatever order you prefer:

`%Y` Four digit year.

`%y` Two digit year.

`%m` Two digit month.

`%d` Two digit date.

So `"%Y%m%d"` yields '`20111214`' which provides a date that is simple to sort on.

12.5.5.2 Weekdays

You can have additional weekday information in your date with '`%A`' as

`%m-%d-%Y %A`
yields '`02-10-2010 Wednesday`'.

`%A %m-%d-%Y`
: yields '`Wednesday 02-10-2010`'.

These are options you can select for weekday

`%a` weekday, abbreviated Wed.

`%A` weekday, full Wednesday.

`%d` day of the month (dd), zero padded up to 10.

`%e` day of the month (dd), no leading zero up to 10.

`%j` day of year, zero padded 000–366.

`%u` day of week starting with Monday (1), i.e. `mtwtfss` 3.

`%w` day of week starting with Sunday (0), i.e. `smtwtfs` 3.

12.5.5.3 Month

You can have additional month information in your date with '`%B`' as

`%m-%d-%Y %B`
: yields '`02-10-2010 February`'.

`%B %m-%d-%Y`
: yields '`February 02-10-2010`'.

These are options you can select for month

`%m` '`mm`' month as two digits.

`%b` Locale's abbreviated month, for example '`02`' might be abbreviated as '`Feb`'.

`%B` Locale's full month, variable length, e.g. February.

12.5.5.4 Miscellaneous Date Codes

Additional date format parameters which can be used:

`%U` week number Sunday as first day of week, ranging 01–53.

`%W` week number Monday as first day of week, ranging 01–53.

`%V` week of the year, ranging 01–53.

`%C` century, ranging 00–99.

`%D` yields `%m/%d/%y` as in '`02/10/10`'.

`%x` locale's date representation, as '`02/10/2010`' for the U.S.

`%F` yields `%Y-%m-%d` as in '`2010-02-10`'.

12.5.6 Text Formatting

The following format functions allow you limited formatting of text:

`ansify_if(value, color)`
: Surrounds the string representing value with ANSI codes to give it `color` on an TTY display. Has no effect if directed to a file.

`justify(value, first_width, latter_width, right_justify, colorize)`
: Right or left justify the string representing `value`. The width of the field in the first line is given by `first_width`. For subsequent lines the width is given by `latterwidth`. If `latter_width=-1`, then `first_width` is use for all lines. If `right_justify=true` then the field is right justify within the width of the field. If it is `false`, then the field is left justified and padded to the full width of the field. If `colorize` is true, then ledger will honor color settings.

`join(STR)`
: Replaces line feeds in `STR` with '`\n`'.

`quoted(STR)`
: Return `STR` surrounded by double quotes, '`"STR"`'.

`strip(value)`
: Values can have numerous annotations, such as effective dates and lot prices. `strip` removes these annotations.

12.5.7 Data File Parsing Information

The following format strings provide locational metadata regarding the coordinates of entries in the source data file(s) that generated the posting.

`filename` the name of ledger the data file from whence the posting came, abbreviated '`S`'.

`beg_pos` character position in `filename` where entry for posting begins, abbreviated '`B`'.

`end_pos` character position in `filename` where entry for posting ends, abbreviated '`E`'.

`beg_line` line number in `filename` where entry for posting begins, abbreviated '`b`'.

`end_line` line number in `filename` where posting's entry for posting ends, abbreviated '`e`'.

13 Extending with Python

Python can be used to extend your Ledger experience. But first, a word must be said about Ledger's data model, so that other things make sense later.

13.1 Basic data traversal

Every interaction with Ledger happens in the context of a Session. Even if you don't create a session manually, one is created for you by the top-level interface functions. The Session is where objects live like the Commodities that Amounts refer to.

The make a Session useful, you must read a Journal into it, using the function '`read_journal`'. This reads Ledger data from the given file, populates a Journal object within the current Session, and returns a reference to that Journal object.

Within the Journal live all the Transactions, Postings, and other objects related to your data. There are also AutomatedTransactions and PeriodicTransactions, etc.

Here is how you would traverse all the postings in your data file:

```
import ledger

for xact in ledger.read_journal("sample.dat").xacts():
    for post in xact.posts():
        print "Transferring %s to/from %s" % (post.amount, post.account)
```

13.2 Raw versus Cooked

Ledger data exists in one of two forms: raw and cooked. Raw objects are what you get from a traversal like the above, and represent exactly what was seen in the data file. Consider this journal:

```
= true
    (Assets:Cash)    $100

2012-03-01 KFC
    Expenses:Food    $100
    Assets:Credit
```

In this case, the *raw* regular transaction in this file is:

```
2012-03-01 KFC
    Expenses:Food    $100
    Assets:Credit
```

While the *cooked* form is:

```
2012-03-01 KFC
    Expenses:Food    $100
    Assets:Credit   $-100
    (Assets:Cash)    $100
```

So the easy way to think about raw vs. cooked is that raw is the unprocessed data, and cooked has had all considerations applied.

When you traverse a Journal by iterating over its transactions, you are generally looking at raw data. In order to look at cooked data, you must generate a report of some kind by querying the journal:

```
for post in ledger.read_journal("sample.dat").query("food"):
    print "Transferring %s to/from %s" % (post.amount, post.account)
```

The reason why queries iterate over postings instead of transactions is that queries often return only a "slice" of the transactions they apply to. You can always get at a matching posting's transaction by looking at its `xact` member:

```
last_xact = None
for post in ledger.read_journal("sample.dat").query(""):
    if post.xact != last_xact:
        for post in post.xact.posts:
            print "Transferring %s to/from %s" % (post.amount,
            post.account)
        last_xact = post.xact
```

This query ends up reporting every cooked posting in the Journal, but does it transaction-wise. It relies on the fact that an unsorted report returns postings in the exact order they were parsed from the journal file.

13.3 Queries

The Journal.query() method accepts every argument you can specify on the command-line, including `--options`.

Since a query "cooks" the journal it applies to, only one query may be active for that journal at a given time. Once the query object is gone (after the for loop), then the data reverts back to its raw state.

13.4 Embedded Python

You can embed Python into your data files using the 'python' directive:

```
python
    import os
    def check_path(path_value):
        print "%s => %s" % (str(path_value), os.path.isfile(str(path_value)))
        return os.path.isfile(str(path_value))

tag PATH
    assert check_path(value)

2012-02-29 KFC
    ; PATH: somebogusfile.dat
    Expenses:Food                $20
    Assets:Cash
```

Any Python functions you define this way become immediately available as valexpr functions.

13.5 Amounts

When numbers come from Ledger, like post.amount, the type of the value is Amount. It can be used just like an ordinary number, except that addition and subtraction are restricted to amounts with the same commodity. If you need to create sums of multiple commodities, use a Balance. For example:

```
total = Balance()
for post in ledger.read_journal("sample.dat").query(""):
```

```
    total += post.amount
print total
```

14 Ledger for Developers

14.1 Internal Design

Ledger is developed as a tiered set of functionality, where lower tiers know nothing about the higher tiers. In fact, multiple libraries are built during the development the process, and link unit tests to these libraries, so that it is a link error for a lower tier to violate this modularity.

Those tiers are:

- Utility code

  There's lots of general utility in Ledger for doing time parsing, using Boost.Regex, error handling, etc. It's all done in a way that can be reused in other projects as needed.

- Commoditized Amounts (amount_t, commodity_t and friends)

  A numerical abstraction combining multi-precision rational numbers (via GMP) with commodities. These structures can be manipulated like regular numbers in either C++ or Python (as Amount objects).

- Commodity Pool

  Commodities are all owned by a commodity pool, so that future parsing of amounts can link to the same commodity and established a consistent price history and record of formatting details.

- Balances

  Adds the concept of multiple amounts with varying commodities. Supports simple arithmetic, and multiplication and division with non-commoditized values.

- Price history

  Amounts have prices, and these are kept in a data graph which the amount code itself is only dimly aware of (there's three points of access so an amount can query its revalued price on a given date).

- Values

  Often the higher layers in Ledger don't care if something is an amount or a balance, they just want to add stuff to it or print it. For this, I created a type-erasure class, value_t/Value, into which many things can be stuffed and then operated on. They can contain amounts, balances, dates, strings, etc. If you try to apply an operation between two values that makes no sense (like dividing an amount by a balance), an error occurs at runtime, rather than at compile-time (as would happen if you actually tried to divide an `amount_t` by a `balance_t`).

  This is the core data type for the value expression language.

- Value expressions

  The next layer up adds functions and operators around the Value concept. This lets you apply transformations and tests to Values at runtime without having to bake it into C++. The set of functions available is defined by each object type in Ledger (posts, accounts, transactions, etc.), though the core engine knows nothing about these. At its base, it only knows how to apply operators to values, and how to pass them to and receive them from functions.

- Query expressions

  Expressions can be onerous to type at the command-line, so there's a shorthand for reporting called "query expressions". These add no functionality of their own, but are purely translated from the input string (cash) down to the corresponding value expression '`(account =~ /cash/)`'. This is a convenience layer.

- Format strings

  Format strings let you interpolate value expressions into strings, with the requirement that any interpolated value have a string representation. Really all this does is calculate the value expression in the current report context, call the resulting value's `to_string()` method, and stuffs the result into the output string. It also provides printf-like behavior, such as min/max width, right/left justification, etc.

- Journal items

  Next is a base type shared by anything that can appear in a journal: an item_t. It contains details common to all such parsed entities, like what file and line it was found on, etc.

- Journal posts

  The most numerous object found in a Journal, postings are a type of item that contain an account, an amount, a cost, and metadata. There are some other complications, like the account can be marked virtual, the amount could be an expression, etc.

- Journal transactions

  Postings are owned by transactions, always. This subclass of item_t knows about the date, the payee, etc. If a date or metadata tag is requested from a posting and it doesn't have that information, the transaction is queried to see if it can provide it.

- Journal accounts

  Postings are also shared by accounts, though the actual memory is managed by the transaction. Each account knows all the postings within it, but contains relatively little information of its own.

- The Journal object

  Finally, all transactions with their postings, and all accounts, are owned by a `journal_t` object. This is the go-to object for querying ad reporting on your data.

- Textual journal parser

  There is a textual parser, wholly contained in `textual.cc`, which knows how to parse text into journal objects, which then get "finalized" and added to the journal. Finalization is the step that enforces the double-entry guarantee.

- Iterators

  Every journal object is "iterable", and these iterators are defined in `iterators.h` and `iterators.cc`. This iteration logic is kept out of the basic journal objects themselves for the sake of modularity.

- Comparators

  Another abstraction isolated to its own layer, this class encapsulating the comparison of journal objects, based on whatever value expression the user passed to `--sort` *VEXPR*.

- Temporaries

  Many reports bring pseudo-journal objects into existence, like postings which report totals in a '`Total`' account. These objects are created and managed by a `temporaries_t` object, which gets used in many places by the reporting filters.

- Option handling

  There is an option handling subsystem used by many of the layers further down. It makes it relatively easy for me to add new options, and to have those option settings immediately accessible to value expressions.

- Session objects

  Every journal object is owned by a session, with the session providing support for that object. In GUI terms, this is the Controller object for the journal Data object, where every document window would be a separate session. They are all owned by the global scope.

- Report objects

  Every time you create any report output, a report object is created to determine what you want to see. In the Ledger REPL, a new report object is created every time a command is executed. In CLI mode, only one report object ever comes into being, as Ledger immediately exits after displaying the results.

- Reporting filters

  The way Ledger generates data is this: it asks the session for the current journal, and then creates an iterator applied to that journal. The kind of iterator depends on the type of report.

  This iterator is then walked, and every object yielded from the iterator is passed to an "item handler", whose type is directly related to the type of the iterator.

  There are many, many item handlers, which can be chained together. Each one receives an item (post, account, xact, etc.), performs some action on it, and then passes it down to the next handler in the chain. There are filters which compute the running totals; that queue and sort all the input items before playing them back out in a new order; that filter out items which fail to match a predicate, etc. Almost every reporting feature in Ledger is related to one or more filters. Looking at `filters.h`, there are over 25 of them defined currently.

- The filter chain

  How filters get wired up, and in what order, is a complex process based on all the various options specified by the user. This is the job of the chain logic, found entirely in `chain.cc`. It took a really long time to get this logic exactly right, which is why I haven't exposed this layer to the Python bridge yet.

- Output modules

  Although filters are great and all, in the end you want to see stuff. This is the job of special "leaf" filters called output modules. They are implemented just like a regular filter, but they don't have a "next" filter to pass the data on down to. Instead, they are the end of the line and must do something with the item that results in the user seeing something on their screen or in a file.

- Select queries

  Select queries know a lot about everything, even though they implement their logic by implementing the user's query in terms of all the other features thus presented. Select queries have no functionality of their own, they are simple a shorthand to provide access to much of Ledger's functionality via a cleaner, more consistent syntax.

- The Global Scope

  There is a master object which owns every other objects, and this is Ledger's global scope. It creates the other objects, provides REPL behavior for the command-line utility, etc. In GUI terms, this is the Application object.

- The Main Driver

  This creates the global scope object, performs error reporting, and handles command-line options which must precede even the creation of the global scope, such as `--debug` *`CODE`*.

And that's Ledger in a nutshell. All the rest are details, such as which value expressions each journal item exposes, how many filters currently exist, which options the report and session scopes define, etc.

14.2 Journal File Format for Developers

This chapter offers a complete description of the journal data format, suitable for implementers in other languages to follow. For users, the chapter on keeping a journal is less extensive, but more typical of common usage (see Chapter 4 [Keeping a Journal], page 17).

Data is collected in the form of *transactions* which occur in one or more *journal files*. Each transaction, in turn, is made up of one or more *postings*, which describe how *amounts* flow from one *account* to another. Here is an example of the simplest of journal files:

```
2010/05/31 Just an example
    Expenses:Some:Account                $100.00
    Income:Another:Account
```

In this example, there is a transaction date, a payee, or description of the transaction, and two postings. The postings show movement of one hundred dollars from an account within the Income hierarchy, to the specified expense account. The name and meaning of these accounts is arbitrary, with no preferences implied, although you will find it useful to follow standard accounting practices (see Chapter 3 [Principles of Accounting with Ledger], page 7).

Since an amount is missing from the second posting, it is assumed to be the inverse of the first. This guarantees the cardinal rule of double-entry accounting: the sum of every transaction must balance to zero, or it is in error. Whenever Ledger encounters a *null posting* in a transaction, it uses it to balance the remainder.

It is also typical, though not enforced, to think of the first posting as the destination, and the final as the source. Thus, the amount of the first posting is typically positive. Consider:

```
2010/05/31 An income transaction
    Assets:Checking       $1,000.00
    Income:Salary

2010/05/31 An expense transaction
```

```
    Expenses:Dining          $100.00
    Assets:Checking
```

14.2.1 Comments and meta-data

Comments are generally started using a '`;`'. However, in order to increase compatibility with other text manipulation programs and methods three additional comment characters are valid if used at the beginning of a line: '`#`', '`|`', and '`*`'.

14.2.2 Specifying Amounts

The heart of a journal is the amounts it records, and this fact is reflected in the diversity of amount expressions allowed. All of them are covered here, though it must be said that sometimes, there are multiple ways to achieve a desired result.

Note: It is important to note that there must be at least two spaces between the end of the account and the beginning of the amount (including a commodity designator).

14.2.2.1 Integer Amounts

In the simplest form, bare decimal numbers are accepted:

```
2010/05/31 An income transaction
    Assets:Checking        1000.00
    Income:Salary
```

Such amounts may only use an optional period for a decimal point. These are referred to as *integer amounts* or *uncommoditized amounts*. In most ways they are similar to *commoditized amounts*, but for one significant difference: They always display in reports with *full precision*. More on this in a moment. For now, a word must be said about how Ledger stores numbers.

Every number parsed by Ledger is stored internally as an infinite-precision rational value. Floating-point math is never used, as it cannot be trusted to maintain precision of values. So, in the case of '`1000.00`' above, the internal value is '`100000/100`'.

While rational numbers are great at not losing precision, the question arises: How should they be displayed? A number like '`100000/100`' is no problem, since it represents a clean decimal fraction. But what about when the number '`1/1`' is divided by three? How should one print '`1/3`', an infinitely repeating decimal?

Ledger gets around this problem by rendering rationals into decimal at the last possible moment, and only for display. As such, some rounding must, at times, occur. If this rounding would affect the calculation of a running total, special accommodation postings are generated to make you aware it has happened. In practice, it happens rarely, but even then it does not reflect adjustment of the *internal amount*, only the displayed amount.

What has still not been answered is how Ledger rounds values. Should '`1/3`' be printed as '`0.33`' or '`0.33333`'? For commoditized amounts, the number of decimal places is decided by observing how each commodity is used; but in the case of integer amounts, an arbitrary factor must be chosen. Initially, this factor is six. Thus, '`1/3`' is printed back as '`0.333333`'. Further, this rounding factor becomes associated with each particular value, and is carried through mathematical operations. For example, if that particular number were multiplied by itself, the decimal precision of the result would be twelve. Addition and subtraction do not affect precision.

Since each integer amount retains its own display precision, this is called *full precision*, as opposed to commoditized amounts, which always look to their commodity to know what precision they should round to, and so use *commodity precision*.

14.2.2.2 Commoditized Amounts

A *commoditized amount* is an integer amount which has an associated commodity. This commodity can appear before or after the amount, and may or may not be separated from it by a space. Most characters are allowed in a commodity name, except for the following:

- Any kind of white-space
- Numerical digits
- Punctuation: `.,;:?!`
- Mathematical and logical operators: `-+*/^&|=`
- Bracketing characters: `<>[](){}`
- The at symbol: `@`

And yet, any of these may appear in a commodity name if it is surrounded by double quotes, for example:

```
100 "EUN+133"
```

If a *quoted commodity* is found, it is displayed in quotes as well, to avoid any confusion as to which part is the amount, and which part is the commodity.

Another feature of commoditized amounts is that they are reported back in the same form as parsed. If you specify dollar amounts using '`$100`', they will print the same; likewise with '`100 $`' or '`$100.000`'. You may even use decimal commas, such as '`$100,00`', or thousand-marks, as in '`$10,000.00`'.

These display characteristics become associated with the commodity, with the result being that all amounts of the same commodity are reported consistently. Where this is most noticeable is the *display precision*, which is determined by the most precise value seen for a given commodity—in most cases.

Ledger makes a distinction between *observed amounts* and unobserved amounts. An observed amount is critiqued by Ledger to determine how amounts using that commodity should be displayed; unobserved amounts are significant in their value only—no matter how they are specified, it does not change how other amounts in that commodity will be displayed.

An example of this is found in cost expressions, covered next.

14.2.3 Posting costs

You have seen how to specify either a commoditized or an integer amount for a posting. But what if the amount you paid for something was in one commodity, and the amount received was another? There are two main ways to express this:

```
2010/05/31 Farmer's Market
    Assets:My Larder           100 apples
    Assets:Checking                -$20.00
```

In this example, you have paid twenty dollars for one hundred apples. The cost to you is twenty cents per apple, and Ledger calculates this implied cost for you. You can also make the cost explicit using a *cost amount*:

```
2010/05/31 Farmer's Market
    Assets:My Larder           100 apples @ $0.200000
    Assets:Checking
```

Here the *per-unit cost* is given explicitly in the form of a cost amount; and since cost amounts are *unobserved*, the use of six decimal places has no effect on how dollar amounts are displayed in the final report. You can also specify the *total cost*:

```
2010/05/31 Farmer's Market
    Assets:My Larder           100 apples @@ $20
    Assets:Checking
```

These three forms have identical meaning. In most cases the first is preferred, but the second two are necessary when more than two postings are involved:

```
2010/05/31 Farmer's Market
    Assets:My Larder           100 apples        @ $0.200000
    Assets:My Larder           100 pineapples    @ $0.33
    Assets:My Larder           100 "crab apples" @ $0.04
    Assets:Checking
```

Here the implied cost is '`$57.00`', which is entered into the null posting automatically so that the transaction balances.

14.2.4 Primary commodities

In every transaction involving more than one commodity, there is always one which is the *primary commodity*. This commodity should be thought of as the exchange commodity, or the commodity used to buy and sell units of the other commodity. In the fruit examples above, dollars are the primary commodity. This is decided by Ledger based on the placement of the commodity in the transaction:

```
2010/05/31 Sample Transaction
    Expenses               100 secondary
    Assets                  -50 primary

2010/05/31 Sample Transaction
    Expenses               100 secondary @ 0.5 primary
    Assets

2010/05/31 Sample Transaction
    Expenses               100 secondary @@ 50 primary
    Assets
```

The only case where knowledge of primary versus secondary comes into play is in reports that use the `--market (-V)` or `--basis (-B)` options. With these, only primary commodities are shown.

If a transaction uses only one commodity, this commodity is also considered a primary. In fact, when Ledger goes about ensuring that all transactions balance to zero, it only ever asks this of primary commodities.

14.3 Developer Commands

14.3.1 `echo`

This command simply echoes its argument back to the output.

14.3.2 `reload`

Forces ledger to reload any journal files. This function exists to support external programs controlling a running ledger process and does nothing for a command line user.

14.3.3 `source`

The `source` command takes a journal file as an argument and parses it checking for errors; no other reports are generated, and no other arguments are necessary. Ledger will return success if no errors are found.

14.3.4 Debug Options

These options are primarily for Ledger developers, but may be of some use to a user trying something new.

`--args-only`
: Ignore init files and environment variables for the ledger run.

`--debug` *CODE*
: If Ledger has been built with debug options this will provide extra data during the run. The following are the available *CODES* to debug:

```
account.display            expr.calc.when                  org.next_amount
accounts.sorted            expr.compile                    org.next_total
amount.convert             filters.changed_value           parser.error
amount.is_zero             filters.changed_value.rounding  pool.commodities
amount.parse               filters.collapse                post.assign
amount.price               filters.forecast                python.init
amount.truncate            filters.revalued                python.interp
amount.unround             format.abbrev                   query.mask
amounts.commodities        format.expr                     report.predicate
amounts.refs               generate.post                   scope.symbols
archive.journal            generate.post.string            textual.include
auto.columns               item.meta                       textual.parse
budget.generate            ledger.read                     timelog
commodity.annotated.strip  ledger.validate                 times.epoch
commodity.annotations      lookup                          times.interval
commodity.compare          lookup.account                  times.parse
commodity.download         mask.match                      value.sort
commodity.prices.add       memory.counts                   value.storage.refcount
commodity.prices.find      memory.counts.live              xact.extend
convert.csv                memory.debug                    xact.extend.cleared
csv.mappings               op.cons                         xact.extend.fail
csv.parse                  op.memory                       xact.finalize
draft.xact                 option.args
expr.calc                  option.names
```

`--trace` *INT*
: Enable tracing. The *INT* specifies the level of trace desired:

```
LOG_OFF                    0
LOG_CRIT                   1
```

LOG_FATAL	2
LOG_ASSERT	3
LOG_ERROR	4
LOG_VERIFY	5
LOG_WARN	6
LOG_INFO	7
LOG_EXCEPT	8
LOG_DEBUG	9
LOG_TRACE	10
LOG_ALL	11

`--verbose`
: Print detailed information on the execution of Ledger.

`--verify` Enable additional assertions during run-time. This causes a significant slowdown. When combined with `--debug` *`CODE`* ledger will produce memory trace information.

`--verify-memory`
: FIX THIS ENTRY

`--version`
: Print version information and exit.

14.3.5 Pre-Commands

Pre-commands are useful when you aren't sure how a command or option will work. The difference between a pre-command and a regular command is that pre-commands ignore the journal data file completely, nor is the user's init file read.

`eval` *`VEXPR`*
: Evaluate the given value expression against the model transaction.

`format` *`FORMAT_STRING`*
: Print details of how ledger uses the given formatting description and apply it against a model transaction.

`generate` Randomly generates syntactically valid Ledger data from a seed. Used by the '`GenerateTests`' harness for development testing.

`parse` *`VEXPR`*
`expr` *`VEXPR`*
: Print details of how ledger uses the given value expression description and apply it against a model transaction.

`period` *`PERIOD_EXPRESSION`*
: Evaluate the given period and report how Ledger interprets it:

```
$ ledger period "this year"
--- Period expression tokens ---
TOK_THIS: this
TOK_YEAR: year
END_REACHED: <EOF>

--- Before stabilization ---
```

```
   range: in year 2011

--- After stabilization ---
   range: in year 2011
   start: 11-Jan-01
  finish: 12-Jan-01

--- Sample dates in range (max. 20) ---
 1: 11-Jan-01
```

`query`
`args` Evaluate the given arguments and report how Ledger interprets it against the following model transaction:

```
$ ledger query "/Book/"
--- Input arguments ---
("/Book/")

--- Context is first posting of the following transaction ---
2004/05/27 Book Store
    ; This note applies to all postings. :SecondTag:
    Expenses:Books                 20 BOOK @ $10
    ; Metadata: Some Value
    ; Typed:: $100 + $200
    ; :ExampleTag:
    ; Here follows a note describing the posting.
    Liabilities:MasterCard        $-200.00

--- Input expression ---
(account =~ /Book/)

--- Text as parsed ---
(account =~ /Book/)

--- Expression tree ---
0x7fd639c0da40    O_MATCH (1)
0x7fd639c10170     IDENT: account (1)
0x7fd639c10780     VALUE: /Book/ (1)

--- Compiled tree ---
0x7fd639c10520    O_MATCH (1)
0x7fd639c0d6c0     IDENT: account (1)
0x7fd639c0d680      FUNCTION (1)
0x7fd639c10780     VALUE: /Book/ (1)

--- Calculated value ---
true
```

`script` FIX THIS ENTRY

`template` Shows the insertion template that the `xact` sub-command generates. This is a debugging command.

14.4 Ledger Development Environment

14.4.1 `acprep` build configuration tool

14.4.2 Testing Framework

Ledger source ships with a fairly complete set of tests to verify that all is well, and no old errors have resurfaced. Tests are run individually with `ctest`. All tests can be run using `make check` or `ninja check` depending on which build tool you prefer.

Once built, the ledger executable resides under the `build` subdirectory in the source tree. Tests are built and stored in the `test` subdirectory for the build. For example, `~/ledger/build/ledger/opt/test`.

14.4.2.1 Running Tests

The complete test suite can be run from the build directory using the check option for the build tool you use. For example, `make check`. The entire test suit lasts around a minute for the optimized built and many times longer for the debug version. While developing and debugging, running individual tests can save a great deal of time.

Individual tests can be run from the `test` subdirectory of the build location. To execute a single test use `ctest -V -R regex`, where the regex matches the name of the test you want to build.

There are nearly 300 tests stored under the `test` subdirectory in the main source distribution. They are broken into two broad categories, baseline and regression. To run the `5FBF2ED8` test, for example, issue '`ctest -V -R "5FB"`'.

14.4.2.2 Writing Tests

15 Major Changes from version 2.6

The following have been removed from Ledger 3.0:

- OFX support.
- GnuCash file import.
- The option `--performance (-g)`.
- The balance report now defaults to showing all relevant accounts. This is the opposite of 2.x. That is, `bal` in 3.0 does what '`-s bal`' did in 2.x. To see 2.6 behavior, use `--collapse (-n)` option in 3.0, like '`bal -n`'. The `--subtotal (-s)` option no longer has any effect on balance reports.

The following are deprecated in Ledger 3.0:

- Single character value expressions are deprecated and should be changed to the new value expressions available in 3.0
- The following environment variables have been renamed in Ledger 3.0:

  `LEDGER` is now `LEDGER_FILE`,

  `LEDGER_INIT`
  is now `LEDGER_INIT_FILE`,

  `PRICE_HIST`
  is now `LEDGER_PRICE_DB`,

  `PRICE_EXP`
  is now `LEDGER_PRICE_EXP`.

Appendix A Example Journal File

The following journal file is included with the source distribution of ledger. It is called `drewr.dat` and exhibits many ledger features, include automatic and virtual transactions,

```
; -*- ledger -*-

= /^Income/
  (Liabilities:Tithe)                   0.12

~ Monthly
  Assets:Checking                     $500.00
  Income:Salary

2003/12/01 * Checking balance
  Assets:Checking                   $1,000.00
  Equity:Opening Balances

2003/12/20 Organic Co-op
  Expenses:Food:Groceries             $ 37.50  ; [=2004/01/01]
  Expenses:Food:Groceries             $ 37.50  ; [=2004/02/01]
  Expenses:Food:Groceries             $ 37.50  ; [=2004/03/01]
  Expenses:Food:Groceries             $ 37.50  ; [=2004/04/01]
  Expenses:Food:Groceries             $ 37.50  ; [=2004/05/01]
  Expenses:Food:Groceries             $ 37.50  ; [=2004/06/01]
  Assets:Checking                   $ -225.00

2003/12/28=2004/01/01 Acme Mortgage
  Liabilities:Mortgage:Principal    $  200.00
  Expenses:Interest:Mortgage        $  500.00
  Expenses:Escrow                   $  300.00
  Assets:Checking                  $ -1000.00

2004/01/02 Grocery Store
  Expenses:Food:Groceries             $ 65.00
  Assets:Checking

2004/01/05 Employer
  Assets:Checking                   $ 2000.00
  Income:Salary

2004/01/14 Bank
  ; Regular monthly savings transfer
  Assets:Savings                     $ 300.00
  Assets:Checking

2004/01/19 Grocery Store
  Expenses:Food:Groceries             $ 44.00
  Assets:Checking

2004/01/25 Bank
  ; Transfer to cover car purchase
  Assets:Checking                  $ 5,500.00
  Assets:Savings
  ; :nobudget:

2004/01/25 Tom's Used Cars
  Expenses:Auto                    $ 5,500.00
  ; :nobudget:
```

```
  Assets:Checking

2004/01/27 Book Store
  Expenses:Books                  $20.00
  Liabilities:MasterCard

2004/02/01 Sale
  Assets:Checking:Business       $ 30.00
  Income:Sales
```

Appendix B Miscellaneous Notes

Various notes from the discussion list that I haven't incorporated in to the main body of the documentation.

B.1 Cookbook

B.1.1 Invoking Ledger

```
$ ledger --group-by "tag('trip')" bal
$ legder reg --sort "tag('foo')" %foo
$ ledger cleared VWCU NFCU Tithe Misentry
$ ledger register Joint --uncleared
$ ledger register Checking  --sort d  -d 'd>[2011/04/01]' until 2011/05/25
```

B.1.2 Ledger Files

```
= /^Income:Taxable/
  (Liabilities:Tithe Owed)    -0.1
= /Noah/
  (Liabilities:Tithe Owed)    -0.1
= /Jonah/
  (Liabilities:Tithe Owed)    -0.1
= /Tithe/
  (Liabilities:Tithe Owed)    -1.0
```

Concepts Index

Commands & Options Index

A

B

C

D

E

F

G

H

L

O

P

Q

R

S

T

X