Ledger: Command-Line Accounting

John Wiegley

Table of Contents

1 Introduction

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 ledger 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 ledger. 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 ledger, so we'll describe double-entry accounting in terms of that.

A checkbook ledger 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 ledger is to know how much money is available to spend. That's really the aim of all ledgers.

What computers add is the ability to walk through these transactions, 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 ledger, 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 ledger, is a ledger that keeps track of all your accounts, not just checking. In such a ledger, you record not only who gets paid—in the case of a debit—but where the money came from. In a checkbook ledger, its assumed that all the money comes from your checking account. But in a general ledger, you write transaction 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 ledger 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 ledger for both. In this general ledger you need to record a payment to Pacific Bell for your monthly phone bill. The cost is $23.00, let's say, and you want to pay it from your checking account. In the general ledger you need to say where the money came from, in addition to where it's going to. The entry might look like this:

```
9/29  BAL  Pacific Bell             $-200.00   $-200.00
           Equity:Opening Balances    $200.00
9/29  BAL  Checking                   $100.00    $100.00
           Equity:Opening Balances   $-100.00
9/29  100  Pacific Bell                $23.00    $223.00
           Checking                   $-23.00     $77.00
```

The first line shows a payment to Pacific Bell for $23.00. Because there is no "balance" in a general ledger—it's always zero—we write in the total balance of all payments to "Pacific Bell", which now is $223.00 (previously the balance was $200.00). This is done by looking at the last entry for "Pacific Bell" in the ledger, adding $23.00 to that amount, and writing

the total in the balance column. And the money came from "Checking"—a withdrawal of $23.00—which leaves the ending balance in "Checking" at $77.00. This is a very manual procedure; but that's where computers come in...

The transaction must balance to $0: $23 went to Pacific Bell, $23 came from Checking. There is nothing left over to be accounted for, since the money has simply moved from one account to another. This is the basis of double-entry accounting: that money never pops in or out of existence; it is always a transaction from one account to another.

Keeping a general ledger is the same as keeping two separate ledgers: 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 ledger books, if you deal with multiple accounts.

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

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

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

```
2004/09/29 Pacific Bell
     Payable:Pacific Bell            $-200.00
     Equity:Opening Balances

2004/09/29 Checking
     Accounts:Checking                $100.00
     Equity:Opening Balances

2004/09/29 Pacific Bell
     Payable:Pacific Bell              $23.00
     Accounts:Checking
```

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

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

1.1 Building the program

Ledger is written in ANSI C++, and should compile on any platform. It depends on the GNU multiprecision integer 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] In some special cases, it automatically balances this entry for you.

1.2 Getting help

If you need help on how to use Ledger, or run into problems, you can just the Ledger mailing list at the following Web address:

`https://lists.sourceforge.net/lists/listinfo/ledger-discuss`

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

2 Running Ledger

Ledger has a very simple command-line interface, named—enticing 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...]
```

Command options must always precede the command word. 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 transactions matching those regular expressions. For the `entry` command, the arguments have a special meaning, described below.

The regular expressions arguments always match the account name that a transaction refers to. To match on the payee of the entry instead, precede the regular expression with '`--`'. 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 -- freddie
```

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

2.1 Usage overview

Before getting into the details of how to run Ledger, it will be easier to introduce the features in the context of their typical usage. To that end, this section presents a series of recipes, gradually introducing all of the command-line features of Ledger.

For the purpose of these examples, assume the environment variable *LEDGER* is set to the file '`sample.dat`' (which is included in the distribution), and that the contents of that file are:

```
= /^Expenses:Books/
  (Liabilities:Taxes)             -0.10

~ Monthly
  Assets:Bank:Checking          $500.00
  Income:Salary

2004/05/01 * Checking balance
  Assets:Bank:Checking        $1,000.00
  Equity:Opening Balances

2004/05/01 * Investment balance
  Assets:Brokerage              50 AAPL  $30.00
  Equity:Opening Balances

2004/05/14 * Pay day
  Assets:Bank:Checking          $500.00
  Income:Salary

2004/05/27 Book Store
  Expenses:Books                 $20.00
  Liabilities:MasterCard
```

```
2004/05/27 (100) Credit card company
  Liabilities:MasterCard         $20.00
  Assets:Bank:Checking
```

This sample file demonstrates a basic principle of accounting which it is recommended you follow: Keep all of your accounts under five parent Assets, Liabilities, Income, Expenses and Equity. It is important to do so in order to make sense out of the following examples.

2.1.1 Checking balances

Ledger has seven basic commands, but by far the most often used are `balance` and `register`. To see a summary balance of all accounts, use:

```
ledger bal
```

`bal` is a short-hand for `balance`. This command prints out the summary totals of the five parent accounts used in '`sample.dat`':

```
             $1,480.00
               50 AAPL  Assets
            $-2,500.00  Equity
                $20.00  Expenses
              $-500.00  Income
                $-2.00  Liabilities
--------------------
            $-1,502.00
               50 AAPL
```

None of the child accounts are shown, just the parent account totals. We can see that in '`Assets`' there is $1,480.00, and 50 shares of Apple stock. There is also a negative grand total. Usually the grand total is zero, which means that all accounts balance[1]. In this case, since the 50 shares of Apple stock cost $1,500.00 dollars, then these two amounts balance each other in the grand total. The extra $2.00 comes from a virtual transaction being added by the automatic entry at the top of the file. The entry is virtual because the account name was surrounded by parentheses in an automatic entry. Automatic entries will be discussed later, but first let's remove the virtual transaction from the balance report by using the '`--real`' option:

```
ledger --real bal
```

Now the report is:

```
             $1,480.00
               50 AAPL  Assets
            $-2,500.00  Equity
                $20.00  Expenses
              $-500.00  Income
--------------------
            $-1,500.00
               50 AAPL
```

Since the liability was a virtual transaction, it has dropped from the report and we see that final total is balanced.

But we only know that it balances because '`sample.dat`' is quite simple, and we happen to know that the 50 shares of Apple stock cost $1,500.00. We can verify that things really

[1] It is impossible for accounts not to balance in ledger; it reports an error if a transaction does not balance

balance by reporting the Apple shares in terms of their cost, instead of their quantity. To do this requires the '`--basis`', or '`-B`', option:

```
ledger --real -B bal
```

This command reports:

```
            $2,980.00  Assets
           $-2,500.00  Equity
               $20.00  Expenses
             $-500.00  Income
```

With the basis cost option, the grand total has disappeared, as it is now zero. The confirms that the cost of everything balances to zero, *which must always be true*. Reporting the real basis cost should never yield a remainder[2].

2.1.1.1 Sub-account balances

The totals reported by the balance command are only the topmost parent accounts. To see the totals of all child accounts as well, use the '`-s`' option:

```
ledger --real -B -s bal
```

This reports:

```
            $2,980.00  Assets
            $1,480.00    Bank:Checking
            $1,500.00    Brokerage
           $-2,500.00  Equity:Opening Balances
               $20.00  Expenses:Books
             $-500.00  Income:Salary
```

This shows that the '`Assets`' total is made up from two child account, but that the total for each of the other accounts comes from one child account.

Sometimes you may have a lot of children, nested very deeply, but only want to report the first two levels. This can be done with a display predicate, using a value expression. In the value expression, `T` represents the reported total, and `l` is the display level for the account:

```
ledger --real -B -d "T&l<=2" bal
```

This reports:

```
            $2,980.00  Assets
            $1,480.00    Bank
            $1,500.00    Brokerage
           $-2,500.00  Equity:Opening Balances
               $20.00  Expenses:Books
             $-500.00  Income:Salary
```

Instead of reporting '`Bank:Checking`' as a child of '`Assets`', it report only '`Bank`', since that account is a nesting level of 2, while '`Checking`' is at level 3.

To review the display predicate used—`T&l<=2`—this rather terse expression means: Display an account only if it has a non-zero total (`T`), and its nesting level is less than or equal to 2 (`l<=2`).

[2] If it ever does, then generated transactions are involved, which can be removed using '`--actual`'

2.1.1.2 Specific account balances

While reporting the totals for all accounts can be useful, most often you will want to check the balance of a specific account or accounts. To do this, put one or more account names after the balance command. Since these names are really regular expressions, you can use partial names if you wish:

```
ledger bal checking
```

Reports:

```
            $1,480.00  Assets:Bank:Checking
```

Any number of names may be used:

```
ledger bal checking broker liab
```

Reports:

```
            $1,480.00  Assets:Bank:Checking
              50 AAPL  Assets:Brokerage
               $-2.00  Liabilities
```

In this case no grand total is reported, because you are asking for specific account balances.

For those comfortable with regular expressions, any Perl regexp is allowed:

```
ledger bal ^assets.*checking ^liab
```

Reports:

```
            $1,480.00  Assets:Bank:Checking
               $-2.00  Liabilities:Taxes
```

2.1.2 The register report

While the `balance` command can be very handy for checking account totals, by far the most powerful of Ledger's reporting tools is the `register` command. In fact, internally both commands use the same logic, but report the results differently: `balance` shows the summary totals, while `register` reports each transaction and how it contributes to that total.

Paradoxically, the most basic form of `register` is almost never used, since it displays every transaction:

```
ledger reg
```

`reg` is a short-hand for `register`. This command reports:

```
2004/05/01 Checking balance     Assets:Bank:Checking      $1,000.00    $1,000.00
                                Equity:Opening Balan..   $-1,000.00            0
2004/05/01 Investment balance   Assets:Brokerage            50 AAPL      50 AAPL
                                Equity:Opening Balan..   $-1,500.00   $-1,500.00
                                                                         50 AAPL
2004/05/14 Pay day              Assets:Bank:Checking        $500.00   $-1,000.00
                                                                         50 AAPL
                                Income:Salary              $-500.00   $-1,500.00
                                                                         50 AAPL
2004/05/27 Book Store           Expenses:Books               $20.00   $-1,480.00
                                                                         50 AAPL
                                Liabilities:MasterCard      $-20.00   $-1,500.00
                                                                         50 AAPL
                                (Liabilities:Taxes)          $-2.00   $-1,502.00
                                                                         50 AAPL
```

```
2004/05/27 Credit card company  Liabilities:MasterCard       $20.00   $-1,482.00
                                                                        50 AAPL
                                Assets:Bank:Checking        $-20.00   $-1,502.00
                                                                        50 AAPL
```

This rather verbose output shows every account transaction in '`sample.dat`', and how it affects the running total. The final total is identical to what we saw with the plain `balance` command. To see how things really balance, we can use '`--real -B`', just as we did with `balance`:

```
ledger --real -B reg
```

Reports:

```
2004/05/01 Checking balance     Assets:Bank:Checking      $1,000.00    $1,000.00
                                Equity:Opening Balan..   $-1,000.00            0
2004/05/01 Investment balance   Assets:Brokerage          $1,500.00    $1,500.00
                                Equity:Opening Balan..   $-1,500.00            0
2004/05/14 Pay day              Assets:Bank:Checking        $500.00      $500.00
                                Income:Salary             $-500.00            0
2004/05/27 Book Store           Expenses:Books              $20.00       $20.00
                                Liabilities:MasterCard      $-20.00            0
2004/05/27 Credit card company  Liabilities:MasterCard       $20.00       $20.00
                                Assets:Bank:Checking        $-20.00            0
```

Here we see that everything balances to zero in the end, as it must.

2.1.2.1 Specific register queries

The most common use of the register command is to summarize transactions based on the account(s) they affect. Using '`sample.dat`' as as example, we could look at all book purchases using:

```
ledger reg books
```

Reports:

```
2004/05/29 Book Store           Expenses:Books              $20.00       $20.00
```

If a double-dash ('`--`') occurs in the list of regular expressions, any following arguments are matched against payee names, instead of account names:

```
ledger reg ^liab -- credit
```

Reports:

```
2004/05/29 Credit card company  Liabilities:MasterCard       $20.00       $20.00
```

There are many reporting options for tailoring which transactions are found, and also how to summarize the various amounts and totals that result. These are plumbed in greater depth below.

2.1.3 Selecting transactions

Although the easiest way to use the register is to report all the transactions affecting a set of accounts, it can often result in more information than you want. To cope with an ever-growing amount of data, there are several options which can help you pinpoint your report to exactly the transactions that interest you most. This is called the "calculation" phase of Ledger. All of its related options are documented under '`--help-calc`'.

2.1.3.1 By date

‘`--current`’(‘`-c`’) displays entries occurring on or before the current date. Any entry recorded for a future date will be ignored, as if it had not been seen. This is useful if you happen to pre-record entries, but still wish to view your balances in terms of what is available today.

‘`--begin DATE`’ (‘`-b DATE`’) limits the report to only those entries occurring on or after *DATE*. The running total in the register will start at zero with the first transaction, even if there are earlier entries.

To limit the display only, but still add earlier transactions to the running total, use the display expression ‘`-d 'd>=[DATE]'`’):

```
ledger --basis -b may -d 'd>=[5/14]' reg ^assets
```

Reports:

```
2004/05/14 Pay day             Assets:Bank:Checking        $500.00    $3,000.00
2004/05/27 Credit card company Assets:Bank:Checking        $-20.00    $2,980.00
```

In this example, the displayed transactions start from ‘`5/14`’, but the calculated total starts from the beginning of ‘`may`’.

‘`--end DATE`’ (‘`-e DATE`’) states when reporting should end, both calculation and display. The ending date is inclusive.

The *DATE* argument to the ‘`-b`’ and ‘`-e`’ options can be rather flexible. Assuming the current date to be November 15, 2004, then all of the following are equivalent:

```
ledger -b oct bal
ledger -b "this oct" bal
ledger -b 2004/10 bal
ledger -b 10 bal
ledger -b last bal
ledger -b "last month" bal
```

To constrain the report to a specific time period, use ‘`--period`’ (‘`-p`’). A time period may have both a beginning and an end, or neither, as well as a specified interval. Here are a few examples:

```
ledger -p 2004 bal
ledger -p august bal
ledger -p "from aug to oct" bal
ledger -p "daily from 8/1 to 8/15" bal
ledger -p "weekly since august" bal
ledger -p "monthly from feb to oct" bal
ledger -p "quarterly in 2004" bal
ledger -p yearly bal
```

See Section 2.6 [Period expressions], page 28 for more on syntax. Also, all of the options ‘`-b`’, ‘`-e`’ and ‘`-p`’ may be used together, but whatever information occurs last takes priority. An example of such usage (in a script, perhaps) would be:

```
ledger -b 2004 -e 2005 -p monthly reg ^expenses
```

This command is identical to:

```
ledger -p "monthly in 2004" reg ^expenses
```

The transactions within a period may be sorted using '`--period-sort`', which takes a value expression. This is similar to the '`--sort`' option, except that it sorts within each period entry, rather than sorting all transactions in the report. See the documentation on '`--sort`' below for more details.

2.1.3.2 By status

By default, all regular transactions are included in each report. To limit the report to certain kinds of transactions, use one or more of the following options:

'`-C, --cleared`'
: Consider only cleared transactions.

'`-U, --uncleared`'
: Consider only uncleared and pending transactions.

'`-R, --real`'
: Consider only real (non-virtual) transactions.

'`-L, --actual`'
: Consider only actual (non-automated) transactions.

Cleared transactions are indicated by an asterix placed just before the payee name in a transaction. The meaning of this flag is up to the user, but typically it means that an entry has been seen on a financial statement. Pending transactions use an exclamation mark in the same position, but are mainly used only by reconciling software. Uncleared transactions are for things like uncashed checks, credit charges that haven't appeared on a statement yet, etc.

Real transactions are all non-virtual transactions, where the account name is not surrounded by parentheses or square brackets. Virtual transactions are useful for showing a transfer of money that never really happened, like money set aside for savings without actually transferring it from the parent account.

Actual transactions are those not generated, either as part of an automated entry, or a budget or forecast report. A useful of when you might like to filter out generated transactions is with a budget:

```
ledger --budget --actual reg ^expenses
```

This command outputs all transactions affecting a budgeted account, but without subtracting the budget amount (because the generated transactions are suppressed with '`--actual`'). The report shows how much you actually spent on budgeted items.

2.1.3.3 By relationship

Normally, a register report includes only the transactions that match the regular expressions specified after the command word. For example, to report all expenses:

```
ledger reg ^expenses
```

This reports:

```
2004/05/29 Book Store          Expenses:Books              $20.00       $20.00
```

Using '`--related`' ('`-r`') reports the transactions that did not match your query, but only in entries that otherwise would have matched. This has the effect of indicating where money came from, or when to:

```
ledger -r reg ^expenses
```

Reports:

```
2004/05/29 Book Store          Liabilities:MasterCard       $20.00       $20.00
```

2.1.3.4 By budget

There is more information about budgeting and forecasting in Section 2.9 [Budgeting and forecasting], page 33. Basically, if you have any period entries in your ledger file, you can use these options. A period entry looks like:

```
~ Monthly
  Assets:Bank:Checking          $500.00
  Income:Salary
```

The difference from a regular entry is that the first line begins with a tilde (`~`), and instead of a payee there's a period expression (Section 2.6 [Period expressions], page 28). Otherwise, a period entry is in every other way the same as a regular entry.

With such an entry in your ledger file, the '`--budget`' option will report only transactions that match a budgeted account. Using '`sample.dat`' from above:

```
ledger --budget reg ^income
```

Reports:

```
2004/05/01 Budget entry        Income:Salary              $500.00      $500.00
2004/05/14 Pay day             Income:Salary             $-500.00            0
```

The final total is zero, indicating that the budget matched exactly for the reported period. Budgeting is most often helpful with period reporting; for example, to show monthly budget results use '`--budget -p monthly`'.

The '`--add-budget`' option reports all matching transactions in addition to budget transactions; while '`--unbudgeted`' shows only those that don't match a budgeted account. To summarize:

'`--budget`'
: Show transactions matching budgeted accounts.

'`--unbudgeted`'
: Show transactions matching unbudgeted accounts.

'`--add-budget`'
: Show both budgeted and unbudgeted transactions together (i.e., add the generated budget transactions to the regular report).

A report with the '`--forecast`' option will add budgeted transactions while the specified value expression is true. For example:

```
ledger --forecast 'd<[2005] reg ^income
```

Reports:

```
2004/05/14 Pay day             Income:Salary             $-500.00     $-500.00
2004/12/01 Forecast entry      Income:Salary             $-500.00   $-1,000.00
2005/01/01 Forecast entry      Income:Salary             $-500.00   $-1,500.00
```

The date this report was made was November 5, 2004; the reason the first forecast entry is in december is that forecast entries are only added for the future, and they only stop after the value expression has matched at least once, which is why the January entry appears. A

forecast report can be very useful for determining when money will run out in an account, or for projecting future cash flow:

```
ledger --forecast 'd<[2008]' -p yearly reg ^inc ^exp
```

This reports balances projected income against projected expenses, showing the resulting total in yearly intervals until 2008. For the case of '`sample.dat`', which has no budgeted expenses, the result of the above command (in November 2004) is:

```
2004/01/01 - 2004/12/31    Income:Salary          $-1,000.00   $-1,000.00
                           Expenses:Books             $20.00     $-980.00
2005/01/01 - 2005/12/31    Income:Salary          $-6,000.00   $-6,980.00
2006/01/01 - 2006/12/31    Income:Salary          $-6,000.00  $-12,980.00
2007/01/01 - 2007/12/31    Income:Salary          $-6,000.00  $-18,980.00
2008/01/01 - 2008/01/01    Income:Salary            $-500.00  $-19,480.00
```

2.1.3.5 By value expression

Value expressions can be quite complex, and are treated more fully in Section 2.5 [Value expressions], page 26. They can be used for limiting a report with '`--limit`' ('`-l`'). The following command report income since august, but expenses since october:

```
ledger -l '(/income/&d>=[aug])|(/expenses/&d>=[oct])' reg
```

The basic form of this value expression is '`(A&B)|(A&B)`'. The '`A`' in each part matches against an account name with '`/name/`', while each '`B`' part compares the date of the transaction ('`d`') with a specified month. The resulting report will contain only transactions which match the value expression.

Another use of value expressions is to calculate the amount reported for each line of a register report, or for computing the subtotal of each account shown in a balance report. This example divides each transaction amount by two:

```
ledger -t 'a/2' reg ^exp
```

The '`-t`' option doesn't affect the running total, only how the transaction amount is displayed. To change the running total, use '`-T`'. In that case, you will likely want to use the total ('`O`') instead of the amount ('`a`'):

```
ledger -T 'O/2' reg ^exp
```

2.1.4 Massaging register output

Even after filtering down your data to just the transactions you're interested in, the default reporting method of one transaction per line is often still too much. To combat this complexity, it is possible to ask Ledger to report the details to you in many different forms, summarized in various ways. This is the "display" phase of Ledger, and is documented under '`--help-disp`'.

2.1.4.1 Summarizing

When multiple transactions relate to a single entry, they are reported as part of that entry. For example, in the case of '`sample.dat`':

```
ledger reg -- book
```

Reports:

```
2004/05/29 Book Store      Expenses:Books          $20.00       $20.00
                           Liabilities:MasterCard  $-20.00           0
```

```
                                (Liabilities:Taxes)          $-2.00       $-2.00
```

All three transactions are part of one entry, and as such the entry details are printed only once. To report every entry on a single line, use '`-n`' to collapse entries with multiple transactions:

```
ledger -n reg -- book
```

Reports:

```
2004/05/29 Book Store           <Total>                      $-2.00       $-2.00
```

In the balance report, '`-n`' causes the grand total not to be displayed at the bottom of the report.

If an account occurs more than once in a report, it is possible to combine them all and report the total per-account, using '`-s`'. For example, this command:

```
ledger -B reg ^assets
```

Reports:

```
2004/05/01 Checking balance     Assets:Bank:Checking      $1,000.00    $1,000.00
2004/05/01 Investment balance   Assets:Brokerage          $1,500.00    $2,500.00
2004/05/14 Pay day              Assets:Bank:Checking        $500.00    $3,000.00
2004/05/27 Credit card company  Assets:Bank:Checking        $-20.00    $2,980.00
```

But if the '`-s`' option is added, the result becomes:

```
2004/05/01 - 2004/05/29         Assets:Bank:Checking      $1,480.00    $1,480.00
                                Assets:Brokerage          $1,500.00    $2,980.00
```

When account subtotaling is used, only one entry is printed, and the date and name reflect the range of the combined transactions.

With '`-P`', transactions relating to the same payee are combined. In this case, the date of the combined entry is that of the latest transaction.

'`-x`' changes the payee name for each transaction to be the same as the commodity it uses. This can be especially useful combined with other options, like '`-P`'. For example:

```
ledger -Px reg ^assets
```

Reports:

```
2004/05/29 $                    Assets:Bank:Checking      $1,480.00    $1,480.00
2004/05/01 AAPL                 Assets:Brokerage            50 AAPL    $1,480.00
                                                                         50 AAPL
```

This reports shows the subtotal for each commodity held, and where it is located. To see the basis cost, or initial investment, add '`-B`'. Applied to the example above:

```
2004/05/29 $                    Assets:Bank:Checking      $1,480.00    $1,480.00
2004/05/01 AAPL                 Assets:Brokerage          $1,500.00    $2,980.00
```

The only other options which affect summarized totals is '`-E`', which works only in the balance report. In this case, it shows matching accounts with a zero a balance, which are ordinarily excluded. This can be useful to see all the accounts involved in a report, even if some have no total.

2.1.4.2 Quick periods

Although the '`-p`' option (also '`--period`') is much more versatile, there are other options to make the most common period reports easier:

'`-W, --weekly`'
: Show weekly sub-totals. Same as '`-p weekly`'.

‘`-M, --monthly`’
: Show monthly sub-totals. Same as ‘`-p monthly`’.

‘`-Y, --yearly`’
: Show yearly sub-totals. Same as ‘`-p yearly`’.

There is one kind of period report cannot be done with ‘`-p`’. This is the ‘`--dow`’, or “days of the week” report, which shows summarized totals for each day of the week. The following examples shows a “day of the week” report of income and expenses:

```
ledger --dow reg ^inc ^exp
```

Reports:

```
2004/05/27 Thursdays           Expenses:Books              $20.00       $20.00
2004/05/14 Fridays             Income:Salary             $-500.00     $-480.00
```

2.1.4.3 Ordering and width

The transactions displayed in a report are shown in the same order as they appear in the ledger file. To change the order and sort a report, use the ‘`--sort`’ option. ‘`--sort`’ takes a value expression to determine the value to sort against, making it possible to sort according to complex criteria. Here are some simple and useful examples:

```
ledger --sort d reg ^exp    # sort by date
ledger --sort t reg ^exp    # sort by amount total
ledger --sort -t reg ^exp   # reverse sort by amount total
ledger --sort Ut reg ^exp   # sort by abs amount total
```

For the balance report, you will want to use ‘`T`’ instead of ‘`t`’:

```
ledger --sort T reg ^exp    # sort by amount total
ledger --sort -T reg ^exp   # reverse sort by amount total
ledger --sort UT reg ^exp   # sort by abs amount total
```

The ‘`--sort`’ options sorts all transactions in a report. If periods are used (such as ‘`--monthly`’), this can get somewhat confusing. In that case, you’ll probably want to sort within periods using ‘`--period-sort`’ instead of ‘`--sort`’.

And if the register seems too cramped, and you have a lot of screen real estate, you can use ‘`-w`’ to format the report within 132 acolumns, instead of 80. You are more likely then to see full payee and account names, as well as properly formatted totals when long-named commodities are used.

If you want only the first or last N entries to be printed—which can be very useful for viewing the last 10 entries in your checking account, while also showing the cumulative balance from all entries—use the ‘`--head`’ and/or ‘`--tail`’ options. The two options may be used simultaneously, for example:

```
ledger --tail 20 reg checking
```

If the output from your command is very long, Ledger can output the data to a pager utility, such as `more` or `less`:

```
ledger --pager /usr/bin/less reg checking
```

2.1.4.4 Averages and percentages

To see the running total changed to a running average, use '`-A`'. The final transaction's total will be the overall average of all displayed transactions. The works in conjunction with period reporting, so that you can see your monthly average expenses with:

```
ledger -AM  reg ^expenses:food
ledger -AMn reg ^expenses
```

This works in the balance report too:

```
ledger -AM  bal ^expenses:food
ledger -AMs bal ^expenses
```

The '`-D`' option changes the running average into a deviation from the running average. This only makes sense in the register report, however.

```
ledger -DM reg ^expenses:food
```

In the balance report only, '`-%`' changes the reported totals into a percentage of the parent account. This kind of report is confusing if negative amounts are involved, and doesn't work at all if multiple commodities occur in an account's history. It has a somewhat limited usefulness, therefore, but in certain cases it can be handy, such as reviewing overall expenses:

```
ledger -%s -S T bal ^expenses
```

2.1.4.5 Reporting total data

Normally in the `xml` report, only transaction amounts are printed. To include the running total under a '`<total>`' tag, use '`--totals`'. This does not affect any other report.

In the register report only, the output can be changed with '`-j`' to show only the date and the amount—without commodities. This only makes sense if a single commodity appears in the report, but can be quite useful for scripting, or passing the data to Gnuplot. To show only the date and running total, use '`-J`'.

2.1.4.6 Display by value expression

With '`-d`' you can decide which transactions (or accounts in the balance report) are displayed, according to a value expression. The computed total is not affected, only the display. This can be very useful for shortening a report without changing the running total:

```
ledger -d 'd>=[last month]' reg checking
```

This command shows the checking account's register, beginning from last month, but with the running total reflecting the entire history of the account.

2.1.4.7 Change report format

When dates are printed in any report, the default format is '`%Y/%m/%d`', which yields dates of the form '`YYYY/mm/dd`'. This can be changed with '`-y`', whose argument is a `strftime` string—see your system's C library documentation for the allowable codes. Mostly you will want to use '`%Y`', '`%m`' and '`%d`', in whatever combination is convenient for your locale.

To change the format of the entire reported line, use '`-F`'. It supports quite a large number of options, which are all documented in Section 2.4 [Format strings], page 24. In addition, each specific kind of report (except for `xml`) can be changed using one of the following options:

‘--balance-format’
balance report. Default:

```
%20T  %2_%-a\n
```

‘--register-format’
register report. Default:

```
%D %-.20P %-.22A %12.66t %12.80T\n%/%32|%-.22A %12.66t %12.80T\n
```

‘--print-format’
print report. Default:

```
%D  %-.35P %-.38A %22.108t %22.132T\n%/%48|%-.38A %22.108t %22.132T\n
```

‘--plot-amount-format’
register report when ‘-j’ (plot amount) is used. Default:

```
%D %(St)\n
```

‘--plot-total-format’
register report when ‘-J’ (plot total) is used. Default:

```
%D %(ST)\n
```

‘--equity-format’
equity report. Default:

```
\n%D %X%C%P\n    %-34A  %12o%n\n%/    %-34A  %12o%n\n
```

‘--prices-format’
prices report. Default:

```
\n%D %X%C%P\n%/    %-34A  %12t\n
```

‘--wide-register-format’
register report when ‘-w’ (wide) is used. Default:

```
%D  %-.35P %-.38A %22.108t %22.132T\n%/%48|%-.38A %22.108t %22.132T\n
```

2.1.5 Standard queries

If your ledger file uses the standard top-level accounts: Assets, Liabilities, Income, Expenses, Equity: then the following queries will enable you to generate some typical accounting reports from your data.

Your *net worth* can be determined by balancing assets against liabilities:

```
ledger bal ^assets ^liab
```

By removing long-term investment and loan accounts, you can see your current net liquidity (or liquid net worth):

```
ledger bal ^assets ^liab -retirement -brokerage -loan
```

Balancing expenses against income yields your *cash flow*, or net profit/loss:

```
ledger bal ^exp ^inc
```

In this case, if the number is positive it means you spent more than you earned during the report period.

The most often used command is the “balance” command:

```
export LEDGER=/home/johnw/doc/ledger.dat
ledger balance
```

Here I’ve set my Ledger environment variable to point to where my ledger file is hiding. Thereafter, I needn’t specify it again.

2.1.6 Reporting balance totals

The balance command prints out the summarized balances of all my top-level accounts, excluding sub-accounts. In order to see the balances for a specific account, just specify a regular expression after the balance command:

```
ledger balance expenses:food
```

This will show all the money that's been spent on food, since the beginning of the ledger. For food spending just this month (September), use:

```
ledger -p sep balance expenses:food
```

Or maybe you want to see all of your assets, in which case the -s (show sub-accounts) option comes in handy:

```
ledger -s balance ^assets
```

To exclude a particular account, use a regular expression with a leading minus sign. The following will show all expenses, but without food spending:

```
ledger balance expenses -food
```

2.1.7 Reporting percentages

There is no built-in way to report transaction amounts or account balances in terms of percentages

2.2 Commands

2.2.1 balance

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

2.2.2 register

The `register` command displays all the transactions occurring in a single account, line by line. The account regexp must be specified as the only argument to this command. If any regexps occur after the required account name, the register will contain only those transactions that match. Very useful for hunting down a particular transaction.

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 '`-j`' or '`-J`' to your register command, in order to plot either the amount or total column, respectively.

2.2.3 print

The `print` command prints out ledger entries 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 regexps, which will cause only those transactions 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.

2.2.4 output

The `output` command is very similar to the `print` command, except that it attempts to replicate the specified ledger file exactly. The format of the command is:

```
ledger -f FILENAME output FILENAME
```

Where '`FILENAME`' is the name of the ledger file to output. The reason for specifying this command is that only entries contained within that file will be output, and not an included entries (as can happen with the `print` command).

2.2.5 xml

The `xml` command outputs results similar to what `print` and `register` display, but as an XML form. This data can then be read in and processed. Use the '`--totals`' option to include the running total with each transaction.

2.2.6 emacs

The `emacs` command outputs 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 transactions
 ...)                   ; list of entries
```

2.2.7 equity

The `equity` command prints out accounts balances as if they were entries. This makes it easy to establish the starting balances for an account, such as when Section 3.8 [Archiving previous years], page 44.

2.2.8 prices

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

There is also a `pricesdb` command which outputs the same information as `prices`, but does in a format that can be parsed by Ledger.

2.2.9 entry

The `entry` commands simplifies the creation of new entries. It works on the principle that 80% of all transactions are variants of earlier transactions. Here's how it works:

Say you currently have this transaction 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 `entry` command you can type:

```
ledger entry 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      $-13.50
```

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

There is a shell script in the distribution's '`scripts`' directory called '`entry`', which simplifies the task of adding a new entry to your ledger. It launches `vi` to confirm that the entry looks appropriate.

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

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

2.3 Options

With all of the reports, command-line options are useful to modify the output generated. These command-line options always occur before the command word. This is done to distinguish options from exclusive regular expressions, which also begin with a dash. The basic form for most commands is:

```
ledger [OPTIONS] COMMAND [REGEXPS...] [-- [REGEXPS...]]
```

The *OPTIONS* and *REGEXPS* expressions are both optional. You could just use '`ledger balance`', without any options—which prints a summary of all accounts. But for more specific reporting, or to change the appearance of the output, options are needed.

2.3.1 Basic options

These are the most basic command options. Most likely, the user will want to set them using Section 2.3.5 [Environment variables], page 24, instead of using actual command-line options:

'`--help`' ('`-h`') prints a summary of all the options, and what they are used for. This can be a handy way to remember which options do what. This help screen is also printed if ledger is run without a command.

'`--version`' ('`-v`') prints 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`') reads FILE as a ledger file. This command may be used multiple times. FILE may also be a list of file names separated by colons. Typically, the environment variable `LEDGER_FILE` is set, rather than using this command-line option.

'`--output FILE`' ('`-o FILE`') redirects 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 transactions, but it may contain option settings. To specify options in the init file, use the same syntax as the command-line. Here's an example init file:

```
--price-db ~/finance/.pricedb

; ~/.ledgerrc ends here
```

Option settings on the command-line or in the environment always take precedence over settings in the init file.

'`--cache FILE`' identifies FILE as the default binary cache file. That is, if the ledger files to be read are specified using the environment variable `LEDGER_FILE`, then whenever a command is finished a binary copy will be written to the specified cache, to speed up the loading time of subsequent queries. This filename can also be given using the environment variable `LEDGER_CACHE`, or by putting the option into your init file. The '`--no-cache`' option causes Ledger to always ignore the binary cache.

'`--account NAME`' ('`-a NAME`') specifies the default account which QIF file transactions are assumed to relate to.

2.3.2 Report filtering

These options change which transactions affect the outcome of a report, in ways other than just using regular expressions:

'`--current`'('`-c`') displays only entries occurring on or before the current date.

'`--begin DATE`' ('`-b DATE`') constrains the report to entries on or after *DATE*. Only entries 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 entry. (Note: This is different from using '`--display`' to constrain what is displayed).

'`--end DATE`' ('`-e DATE`') constrains the report so that entries on or after *DATE* are not considered. The ending date is inclusive.

'`--period STR`' ('`-p STR`') sets the reporting period to *STR*. This will subtotal all matching entries within each period separately, making it easy to see weekly, monthly, quarterly, etc., transaction 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 using period expressions, see Section 2.6 [Period expressions], page 28.

'`--period-sort EXPR`' sorts the transactions 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 -At reg ^Expenses
```

'`--cleared`' ('`-C`') displays only transactions whose entry has been marked "cleared" (by placing an asterix to the right of the date).

'`--uncleared`' ('`-U`') displays only transactions whose entry has not been marked "cleared" (i.e., if there is no asterix to the right of the date).

'`--real`' ('`-R`') displays only real transactions, not virtual. (A virtual transaction is indicated by surrounding the account name with parentheses or brackets; see the section on using virtual transactions for more information).

‘--actual’ (‘-L’) displays only actual transactions, and not those created due to automated transactions.

‘--related’ (‘-r’) displays transactions that are related to whichever transactions 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 entries having a related transaction. For example, if a file had this entry:

```
2004/03/20 Safeway
    Expenses:Food                $65.00
    Expenses:Cash                $20.00
    Assets:Checking             $-85.00
```

And the register command was:

```
ledger -r register food
```

The following would be output, showing the transactions related to the transaction that matched:

```
2004/03/20 Safeway          Expenses:Cash          $-20.00      $-20.00
                            Assets:Checking         $85.00       $65.00
```

‘--budget’ is useful for displaying how close your transactions meet your budget. ‘--add-budget’ also shows unbudgeted transactions, while ‘--unbudgeted’ shows only those. ‘--forecast’ is a related option that projects your budget into the future, showing how it will affect future balances. See Section 2.9 [Budgeting and forecasting], page 33.

‘--limit EXPR’ (‘-l EXPR’) limits which transactions take part in the calculations of a report.

‘--amount EXPR’ (‘-t EXPR’) changes 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 Section 2.5 [Value expressions], page 26.

‘--total EXPR’ (‘-T EXPR’) sets the value expression used for the “totals” column in the register and balance reports.

2.3.3 Output customization

These options affect only the output, but not which transactions are used to create it:

‘--collapse’ (‘-n’) causes entries in a register report with multiple transactions to be collapsed into a single, subtotaled entry.

‘--subtotal’ (‘-s’) causes all entries in a register report to be collapsed into a single, subtotaled entry.

‘--by-payee’ (‘-P’) reports subtotals by payee.

‘--comm-as-payee’ (‘-x’) changes the payee of every transaction to be the commodity used in that transaction. This can be useful when combined with other options, such as ‘-s’.

‘--empty’ (‘-E’) includes even empty accounts in the balance report.

‘--weekly’ (‘-W’) reports transaction totals by the week. The week begins on whichever day of the week begins the month containing that transaction. To set a specific begin date, use a period string, such as ‘weekly from DATE’. ‘--monthly’ (‘-M’) reports transaction totals by month; ‘--yearly’ (‘-Y’) reports transaction totals by year. For more complex period, using the ‘--period’ option described above.

'`--dow`' reports transactions totals for each day of the week. This is an easy way to see if weekend spending is more than on weekdays.

'`--sort EXPR`' ('`-S EXPR`') sorts a report by comparing the values determined using the value expression *EXPR*. For example, using '`-S -UT`' 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 Section 2.5 [Value expressions], page 26.

'`--wide`' ('`-w`') causes the default `register` report to assume 132 columns instead of 80.

'`--head`' causes only the first N entries to be printed. This is different from using the command-line utility `head`, which would limit to the first N transactions. '`--tail`' outputs only the last N entries. 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 entries being printed, for example, it would print all but the first five).

'`--pager`' tells 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`') reports the average transaction value.

'`--deviation`' ('`-D`') reports each transaction's deviation from the average. It is only meaningful in the `register` and `prices` reports.

'`--percentage`' ('`-%`') shows account subtotals in the `balance` report as percentages of the parent account.

'`--totals`' include running total information in the `xml` report.

'`--amount-data`' ('`-j`') changes the `register` report so that it output 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`') changes the `register` report so that it output nothing but the date and totals column, without commodities.

'`--display EXPR`' ('`-d EXPR`') limits which transactions or accounts or 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 transactions, against a running balance which includes all transaction values:

```
ledger -d "d>=[last month]" reg checking
```

The output from this command is very different from the following, whose running total includes only transactions 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 ('`-p`'), or simply a display restricted to the reporting range (using '`-d`').

'`--date-format STR`' ('`-y STR`') changes 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 FORMAT`' in the Ledger initialization file '`~/.ledgerrc`' (or the file referred to by `LEDGER_INIT`).

'`--format STR`' ('`-F STR`') sets the reporting format for whatever report ledger is about to make. See Section 2.4 [Format strings], page 24. There are also specific format commands for each report type:

- '`--balance-format STR`'
- '`--register-format STR`'
- '`--print-format STR`'
- '`--plot-amount-format STR`' (-j `register`)
- '`--plot-total-format STR`' (-J `register`)
- '`--equity-format STR`'
- '`--prices-format STR`'
- '`--wide-register-format STR`' (-w `register`)

2.3.4 Commodity reporting

These options affect how commodity values are displayed:

'`--price-db FILE`' ('`-P FILE`') sets 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, 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
```

'`--price-exp MINS`' ('`-L MINS`') sets the expected freshness of price quotes, in 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`') causes 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 '`-t`' and '`-T`' options. However, there are also several "default" reports, which will satisfy most users basic reporting needs:

`-O, --quantity`
: Reports commodity totals (this is the default)

`-B, --basis`
: Reports the cost basis for all transactions.

`-V, --market`
: Reports the last known market value for all commodities.

`-g, --performance`
: Reports the net gain/loss for each transaction in a `register` report.

`-G --gain` Reports the net gain/loss for all commodities in the report that have a price history.

2.3.5 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 affect 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:

```
--cache /tmp/.mycache
```

2.4 Format strings

Format strings may be used to change the output format of reports. They are specified by passing a formatting string to the '`--format`' ('`-F`') option. Within that string, constructs are allowed which make it possible to display the various parts of an account or transaction in custom ways.

Within a format strings, a substitution is specified using a percent character ('`%`'). The basic format of all substitutions is:

```
%[-][MIN WIDTH][.MAX WIDTH]EXPR
```

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:

```
%-P     An entry's payee, left justified
%20P    The same, right justified, at least 20 chars wide
%.20P   The same, no more than 20 chars wide
%-.20P  Left justified, maximum twenty chars wide
```

The expression following the format constraints can be a single letter, or an expression enclosed in parentheses or brackets. The allowable expressions are:

`%` Inserts a percent sign.

`t` Inserts the results of the value expression specified by '`-t`'. If '`-t`' was not specified, the current report style's value expression is used.

`T` Inserts the results of the value expression specified by '`-T`'. If '`-T`' was not specified, the current report style's value expression is used.

`|` Inserts a single space. This is useful if a width is specified, for inserting a certain number of spaces.

`_` Inserts a space for each level of an account's depth. That is, if an account has two parents, this construct will insert two spaces. If a minimum width is specified, that much space is inserted for each level of depth. Thus '`%5_`', for an account with four parents, will insert twenty spaces.

`(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 transaction's date with a date format string, exactly like those supported by `strftime`. For example: '`%[%Y/%m/%d %H:%M:%S]`'.

`S` Insert the pathname of the file from which the entry's data was read.

`B` Inserts the beginning character position of that entry within the file.

`b` Inserts the beginning line of that entry within the file.

`E` Inserts the ending character position of that entry within the file.

`e` Inserts the ending line of that entry within the file.

`D` By default, this is the same as '`%[%Y/%m%/d]`'. The date format used can be changed at any time with the '`-y`' flag, however. Using '`%D`' gives the user more control over the way dates are output.

`X` If a transaction has been cleared, this inserts '`*`' followed by a space; otherwise nothing is inserted.

`C` Inserts the checking number for an entry, in parentheses, followed by a space; if none was specified, nothing is inserted.

`P` Inserts the payee related to a transaction.

`a` Inserts the optimal short name for an account. This is normally used in balance reports. It prints a parent account's name if that name has not been printed yet, otherwise it just prints the account's name.

`A` Inserts the full name of an account.

`o` Inserts the "optimized" form of a transaction's amount. This is used by the print report. In some cases, this inserts nothing; in others, it inserts the transaction amount and its cost. It's use is not recommend unless you are modifying the print report.

`n` Inserts the note associated with a transaction, preceded by two spaces and a semi-colon, if it exists. Thus, no none becomes an empty string, while the note '`foo`' is substituted as '`  ; foo`'.

`N` Inserts the note associated with a transaction, if one exists.

`/` The '`%/`' construct is special. It separates a format string between what is printed for the first transaction of an entry, and what is printed for all subsequent transactions. If not used, the same format string is used for all transactions.

2.5 Value expressions

Value expressions are an expression language used by Ledger to calculate values used by the program for many different purposes:

1. The values displayed in reports
2. For predicates (where truth is anything non-zero), to determine which transactions are calculated (‘`-l`’) or displayed (‘`-d`’).
3. For sorting criteria, to yield the sort key.
4. In the matching criteria used by automated transactions.

Value expressions support most simple math and logic operators, in addition to a set of one letter functions and variables. A function’s argument is whatever follows it. The following is a display predicate that I use with the `balance` command:

```
ledger -d /^Liabilities/?T<0:UT>100 balance
```

The effect is that account totals are displayed only if: 1) A Liabilities account has a total less than zero; or 2) the absolute value of the account’s total exceeds 100 units of whatever commodity contains. If it contains multiple commodities, only one of them must exceed 100 units.

Display predicates are also very handy with register reports, to constrain which entries are printed. For example, the following command shows only entries from the beginning of the current month, while still calculating the running balance based on all entries:

```
ledger -d "d>[this month]" register checking
```

This advantage to this command’s complexity is that it prints the running total in terms of all entries in the register. The following, simpler command is similar, but totals only the displayed transactions:

```
ledger -b "this month" register checking
```

2.5.1 Variables

Below are the one letter variables available in any value expression. For the register and print commands, these variables relate to individual transactions, and sometimes the account affected by a transaction. 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 ‘`-t`’. In a register report, ‘`-t`’ changes the value column; in a balance report, it has no meaning by default. If ‘`-t`’ was not specified, the current report style’s value expression is used.

`T` This maps to whatever the user specified with ‘`-T`’. In a register report, ‘`-T`’ changes the totals column; in a balance report, this is the value given for each account. If ‘`-T`’ was not specified, the current report style’s value expression is used.

`m` This is always the present moment/date.

2.5.1.1 Transaction/account details

`d` A transaction’s date, as the number of seconds past the epoch. This is always “today” for an account.

`a` The transaction's amount; the balance of an account, without considering children.

`b` The cost of a transaction; the cost of an account, without its children.

`v` The market value of a transaction, or an account without its children.

`g` The net gain (market value minus cost basis), for a transaction 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, it's depth is one.

`n` The index of a transaction, or the count of transactions affecting an account.

`X` 1 if a transaction's entry has been cleared, 0 otherwise.

`R` 1 if a transaction is not virtual, 0 otherwise.

`Z` 1 if a transaction is not automated, 0 otherwise.

2.5.1.2 Calculated totals

`O` The total of all transactions seen so far, or the total of an account and all its children.

`N` The total count of transactions affecting an account and all its children.

`B` The total cost of all transactions seen so far; the total cost of an account and all its children.

`V` The market value of all transactions 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 transactions, or an account and its children. It is the same as '`V-B`'.

2.5.2 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.

2.5.3 Operators

The binary and ternary operators, in order of precedence, are:

1. '`* /`'
2. '`+ -`'
3. '`! < > =`'
4. '`& | ?:`'

2.5.4 Complex expressions

More complicated expressions are possible using:

`NUM` A plain integer represents a commodity-less amount.

`{AMOUNT}` An amount in braces can be any kind of amount supported by ledger, with or without a commodity. Use this for decimal values.

`/REGEXP/`

`W/REGEXP/`
: A regular expression that matches against an account's full name. If a transaction, this will match against the account affected by the transaction.

`//REGEXP/`
`p/REGEXP/`
: A regular expression that matches against an entry's payee name.

`///REGEXP/`
`w/REGEXP/`
: A regular expression that matches against an account's base name. If a transaction, this will match against the account affected by the transaction.

`c/REGEXP/`
: A regular expression that matches against the entry code (the text that occurs between parentheses before the payee name).

`e/REGEXP/`
: A regular expression that matches against a transaction's note, or comment field.

`(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.

`[DATE]` Useful specifying a date in plain terms. For example, you could say '`[2004/06/01]`'.

`@STR(ARGS,...)`
: If Python support is compiled in, this calls the Python function `STR`. It is always be passed at least one argument, of type *ledger.Details*, representing the known account, entry, and transaction details at the time the value expression is computed. Other value expression arguments may also be passed by the user, all of type *Value*.

2.6 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 monthly
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
```

2.7 File 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.

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 an entry. It may be followed by any number of lines, each beginning with whitespace, to denote the entry's account transactions. The format of the first line is:

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

If '`*`' appears after the date, it indicates that entry is "cleared", meaning it has been seen a bank statement, or otherwise verified. If '`!`' appears after the date, it indicates that the entry is "pending"; i.e., tentatively cleared from the user's point of view, but not yet cleared with your financial institution. If a '`CODE`' appears in parentheses, it may be used to indicate a check number, or the type of the transaction. Following these is the payee, or a description of the transaction.

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

After this initial line there should be a set of one or more transactions, just as if it were normal entry. If the amounts of the transactions have no commodity, they will be applied as modifiers to whichever real transaction is matched by the value expression.

`~` A period entry. A period expression must appear after the tilde.

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

`!` A line beginning with an exclamation mark denotes a command directive. It must be immediately followed by the command word. The supported commands are:

'`!include`'
: Include the stated ledger file.

'`!account`'
: The account name is given is taken to be the parent of all transactions that follow, until '`!end`' is seen.

'`!end`' Ends an account block.

'`!python`' If Python support is available, all of the lines following '`!python`' will be passed to the Python interpretor. Any functions defined will

be available to later Python blocks, and can be called from a value expression. The Python code block must be ended with '`!end`'.

`;` A line beginning with a colon indicates a comment, and is ignored.

`Y` If a line begins with a capital Y, it denotes the year used for all subsequent entries that give a date without a year. The year should appear immediately after the Y, for example: '`Y2004`'. This is useful at the beginning of a file, to specify the year for that file. If all entries specify a year, however, this command has no effect.

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

```
P DATE SYMBOL PRICE
```

`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 `entry` 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, o, b, h`
These four relate to timeclock support, which permits ledger to read timelog files. See the timeclock's documentation for more info on the syntax of its timelog files.

2.8 Some typical queries

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

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

From left to right the options mean: Show entries since October, 2003; show all subaccounts; sort by the absolute value of the total; and report the balance for all expenses.

2.8.1 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 t reg ^expenses
```

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

```
ledger -M --period-sort t -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 transactions calculated must match '`^expenses`', while the transactions displayed must match '`mastercard`'. The command would be:

```
ledger -M -r -d /mastercard/ reg ^expenses
```

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

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

```
ledger -s -r -d /mastercard/ reg ^expenses
```

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

2.8.2 Visualizing with Gnuplot

If you have `Gnuplot` installed, you can graph any of the above register reports. The script to do this is included in the ledger distribution, and is named '`scripts/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 '`-j`' or '`-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 -d /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.

2.8.2.1 Typical plots

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-$ transactions

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 ('`-l`') and a display predicate ('`-d`'). The calculation predicates limits the report to transactions whose amount is greater than

$1 (which can only happen if the transaction amount is in dollars). The display predicate limits the entries *displayed* to just those since last February, even those entries from before then will be computed as part of the balance.

2.9 Budgeting and forecasting

2.9.1 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 period entries at the top of your ledger file. A period entry is almost identical to a regular entry, 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 period entries give the usual monthly expenses, as well as one typical yearly expense. For help on finding out what your average monthly expense is for any category, use a command like:

```
ledger -p "this year" -MAs bal ^expenses
```

The reported totals are the current year's average for each account.

Once these period entries 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 -M reg ^exp
```

To see the same report balanced against your budget, use:

```
ledger --budget -M reg ^exp
```

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 -M reg ^exp
```

You can also use these flags with the `balance` command.

2.9.2 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 period entries 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 transactions until the running total is greater than $-500.00. A final transaction is always output, to show 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
```

3 Keeping a ledger

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

The Ledger program aims at making ledger entry as simple as possible. Since it is a command-line tool, it does not provide a user interface for keeping a ledger. If you like, you may use GnuCash to maintain your ledger, in which case the Ledger program 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 ledger, read on. Ledger has been designed to make data entry as simple as possible, by keeping the ledger format easy, and also by automagically determining as much information as possible based on the nature of your entries.

For example, you do not need to tell Ledger about the accounts you use. Any time Ledger sees a transaction involving an account it knows nothing about, it will create it. 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 transaction.

Here is the Pacific Bell example from above, given as a Ledger transaction:

```
9/29 (100) 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 (100) Pacific Bell
    Expenses:Utilities:Phone                  $23.00
    Assets:Checking
```

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

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

In order to write a Ledger entry 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 that when you started your ledger. Make sense?

3.2 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 entry:

```
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
```

Relatedly, 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
```

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 expenses:auto
```

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

3.2.1 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 transaction is best handled with mirrored transactions 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 transactions, so those are what is shown here. First, the personal entry, 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 entry should be immediately followed by an equivalent entry, 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 entry 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 entries 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 entries 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 transactions in separate files, just separate the two entries that were joined above. For example, for both the expense and the pay-back shown above, the following four entries 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:

```
!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
```

(Note: The '`!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 entries, 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.

3.3 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`' option, with the '`-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
```

3.3.1 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 entries 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.

3.3.2 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 transactions, 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:

```
 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 reporetd 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.

3.4 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 for generate more informative reports later on. The more you know, the better analysis you can perform.

If you later sell some of these items to another player, the entry 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.

3.5 Understanding Equity

The most confusing entry 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 an entry to your ledger to reflect this amount. Where will 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 '`-Equity`' at the end of your balance command, to remove the confusing figure from the total.

3.6 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 transactions, 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.

3.7 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 expects to make purchases or reserve 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: 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?

Using a traditional finance package would require that money reside in only one place. But there are really two views to the data: from the account point of view, you want one set of reports; from the fund point of view, another set entirely – and yet both sets should reflect the same expenses and incoming. It's just where the money "resides" that differs.

This situation can be handled using virtual transactions 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 entry ensures that the virtual transactions balance to zero.

Now money can be spent directly from a fund:

```
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 will appear in terms of the funds, income and expenses. In this case, you will likely want to mask out your '`Assets`' account, becausue the balance won't make much sense:

```
ledger bal -^Assets
```

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

```
ledger --real bal
```

If more asset accounts are needed as the source of a transaction, 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
```

3.8 Archiving previous years

After a while, your ledger can get to be pretty large. While this will not slow down the ledger program much—it's designed to process ledger files 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 entries 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* transactions 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.

3.9 Virtual transactions

A virtual transaction is when you, in your mind, see money as moving to a certain place, when in reality that money has not moved at all. There are several scenarios in which this type of tracking comes in handy, and each of them will be discussed in detail.

To enter a virtual transaction, surround the account name in parentheses. This form of usage does not need to balance. However, if you want to ensure the virtual transaction balances with other virtual transactions in the same entry, use square brackets. For example:

```
10/2 Paycheck
    Assets:Checking                    $1000.00
```

```
    Income:Salary                          $-1000.00
    (Debt:Alimony)                           $200.00
```

In this example, after receiving a paycheck an alimony debt is increased—even though no money has moved around yet.

```
10/2  Paycheck
    Assets:Checking                         $1000.00
    Income:Salary                          $-1000.00
    [Savings:Trip]                           $200.00
    [Assets:Checking]                       $-200.00
```

In this example, $200 has been deducted from checking toward savings for a trip. It will appear as though the money has been moved from the account into 'Savings:Trip', although no money has actually moved anywhere.

When balances are displayed, virtual transactions will be factored in. To view balances without any virtual balances factored in, using the '-R' flag, for "reality".

3.10 Automated transactions

As a Bah', I need to compute Huqqu'llh whenever I acquire assets. It is similar to tithing for Jews and Christians, or to Zakt for Muslims. The exact details of computing Huqqu'llh 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 transaction at the top of your ledger file:

```
; This automated entry will compute Huqqu'llh based on this
; journal's transactions.  Any that match will affect the
; Liabilities:Huququ'llah account by 19% of the value of that
; transaction.

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

This automated transaction works by looking at each transaction in the ledger file. If any match the given value expression, 19% of the transaction's value is applied to the 'Liabilities:Huququ'llah' account. So, if $1000 is earned from 'Income:Salary', $190 is added to 'Liabilities:Huqqu'llh'; if $1000 is spent on Rent, $190 is subtracted. The ultimate balance of Huqqu'llh reflects how much is owed in order to fulfill one's obligation to Huqqu'llh. When ready to pay, just write a check to cover the amount shown in 'Liabilities:Huququ'llah'. That entry would look like:

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

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

```
ledger balance Liabilities:Huquq
```

This works fine, but omits one aspect of the law: that Huquq is only due once the liability exceeds the value of 19 mithqls 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" -s bal liab
```

With this command, the current price for gold is downloaded, and the Huqqu'llh 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/?(0/P{2.22 AU}<={-1.0}&0):0" -s bal liab
```

In some cases, you may wish to refer to the account of whatever transaction matched your automated entry'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 transaction, which may be excluded from reports by using '`--real`'.

3.11 Using Emacs to Keep Your Ledger

In the Ledger tarball is an Emacs module, '`ledger.el`'. This module makes the process of keeping a text ledger much easier for Emacs users. I recommend putting this at the top of your ledger file:

```
; -*-ledger-*-
```

And this in your '`.emacs`' file, after copying '`ledger.el`' to your '`site-lisp`' directory:

```
(load "ledger")
```

Now when you edit your ledger file, it will be in `ledger-mode`. `ledger-mode` adds these commands:

C-c C-a For quickly adding new entries based on the form of older ones (see previous section).

C-c C-c Toggles the "cleared" flag of the transaction under point.

C-c C-d Delete the entry under point.

C-c C-r Reconciles an account by displaying the transactions in another buffer, where simply hitting the spacebar will toggle the pending flag of the transaction in the ledger. Once all the appropriate transactions have been marked, press C-c C-c in the reconcile buffer to "commit" the reconciliation, which will mark all of the entries as cleared, and display the new cleared balance in the minibuffer.

C-c C-m Set the default month for new entries added with C-c C-a. This is handy if you have a large number of transactions to enter from a previous month.

C-c C-y Set the default year for new entries added with C-c C-a. This is handy if you have a large number of transactions to enter from a previous year.

Once you enter the reconcile buffer, there are several key commands available:

RET Visit the ledger file entry corresponding to the reconcile entry.

C-c C-c Commit the reconcialation. This marks all of the marked transactions as "cleared", saves the ledger file, and then displays the new cleared balance.

C-l Refresh the reconcile buffer by re-reading transactions from the ledger data file.

SPC Toggle the transaction under point as cleared.

a Add a new entry to the ledger data file, and refresh the reconcile buffer to include its transactions (if the entry is added to the same account as the one being reconciled).

d Delete the entry related to the transaction under point. Note: This may result in multiple transactions being deleted.

n Move to the next line.

p Move to the previous line.

C-c C-r

r Attempt to auto-reconcile the transactions to the entered balance. If it can do so, it will mark all those transactions as pending that would yield the specified balance.

C-x C-s

s Save the ledger data file, and show the current cleared balance for the account being reconciled.

q Quit the reconcile buffer.

There is also an `emacs` command which can be used to output reports in a format directly `read`-able from Emacs Lisp.

3.12 Using GnuCash to Keep Your Ledger

The Ledger tool is fast and simple, but it offers no custom method for actually editing the ledger. It assumes you know how to use a text editor, and like doing so. Perhaps an Emacs mode will appear someday soon to make editing Ledger's data files much easier.

Until then, you are free to use GnuCash to maintain your ledger, and the Ledger program for querying and reporting on the contents of that ledger. It takes a little longer to parse the XML data format that GnuCash uses, but the end result is identical.

Then again, why would anyone use a Gnome-centric, 35 megabyte behemoth to edit their data, and a one megabyte binary to query it?

3.13 Using timeclock to record billable time

The timeclock tool makes it easy to track time events, like clocking into and out of a particular job. These events accumulate in a timelog file.

Each in/out event may have an optional description. If the "in" description is a ledger account name, these in/out pairs may be viewed as virtual transactions, adding time commodities (hours) to that account.

For example, the command-line version of the timeclock tool (which is written in Python) could be used to begin a timelog file like:

```
export TIMELOG=$HOME/.timelog
ti ClientOne category
sleep 10
to waited for ten seconds
```

The '`.timelog`' file now contains:

```
i 2004/10/06 15:21:00 ClientOne  category
o 2004/10/06 15:21:10 waited for ten seconds
```

Ledger parses this directly, as if it had seen the following entry:

```
2004/10/06 category
    (ClientOne)      10s
```

In other words, the timelog event pair is seen as adding 0.00277h (ten seconds) worth of time to the '`ClientOne`' account. This would be considered billable time, which later could be invoiced and credited to accounts receivable:

```
2004/11/01 (INV#1) ClientOne, Inc.
    Receivable:ClientOne       $0.10
    ClientOne              -0.00277h @ $35.00
```

The above transaction converts the clocked time into an invoice for the time spent, at an hourly rate of $35. Once the invoice is paid, the money is deposited from the receivable account into a checking account:

```
2004/12/01 ClientOne, Inc.
    Assets:Checking            $0.10
    Receivable:ClientOne
```

And now the time spent has been turned into hard cash in the checking account.

The advantage to using timeclock and invoicing to bill time is that you will always know, by looking at the balance report, exactly how much unbilled and unpaid time you've spent working for any particular client.

I like to '`!include`' my timelog at the top of my company's accounting ledger, with the attached prefix '`Billable`':

```
; -*-ledger-*-

; This is the ledger file for my company.  But first, include the
; timelog data, entering all of the time events within the umbrella
; account "Billable".

!account Billable
!include /home/johnw/.timelog
!end

; Here follows this fiscal year's transactions for the company.

2004/11/01 (INV#1) ClientOne, Inc.
    Receivable:ClientOne           $0.10
    Billable:ClientOne         -0.00277h @ $35.00

2004/12/01 ClientOne, Inc.
    Assets:Checking                $0.10
    Receivable:ClientOne
```

4 Using XML

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 chapter documents that format.

The general format used for Ledger data is:

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

The data stream is enclosed in a '`ledger`' tag, which contains a series of one or more entries. Each '`entry`' describes the entry and contains a series of one or more transactions:

```
<entry>
  <en:date>2004/03/01</en:date>
  <en:cleared/>
  <en:code>100</en:code>
  <en:payee>John Wiegley</en:payee>
  <en:transactions>
    <transaction>...</transaction>
    <transaction>...</transaction>
    <transaction>...</transaction>...
  </en:transactions>
</entry>
```

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

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

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

```
<transaction>
  <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>
</transaction>
```

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

Lastly follows the amount of the transaction, 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 multi-precision math 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 as long as the '`expat`' library was available when Ledger was built.

5 Extending with Python

Ledger fully supports Python as an extension language. It may be used in a few different forms, which fall into three basic categories:

1. Defining Python functions to use in value expressions
2. Using the ledger library as a Python module
3. Setting up custom initialization using Python

Note that this feature, while functional, is still under development. It will not be documented until it has been fully proven, probably in the next version of ledger. For now, if you wish to make this of this functionality and are willing to debug problems that come up, pass the option '`--enable-python`' to configure, and contact the author via email.

One example of using Python to create a more complex report is in the script file '`scripts/trend`'.