The `bal` manual

`bal` is an extensible tool for keeping accounts in order and studying past spending habits. It is an interactive tool to enter transactions into accounts. This manual gives a basic overview of the package and provides a full reference manual to the various Guile Scheme functions `bal` exposes. I think the writing is bad currently, but hopefully, I will get the time to clean up the writing (I am also accepting editing contributions!).

7 November 2018

1. What `bal` is

bal is an interactive, extensible personal account manager, recording all the good, bad, terrible, and mundane financial decisions you make and helping you plan for the future. It is extensible because its prompt is simply a Guile Scheme interpreter (where you don't enclose the outer S-expression in parenthesis) so it can execute arbitrary code on your transactions. It has a built-in function for making Scheme functions interactive that allows you to specify that the function takes "accounts" or "transactions" (which are simply Scheme lists with certain values) as arguments and have the user select the account or transaction from a menu.

The main interface is a command prompt interface using the Scheme interpreter with some built-in C functions that can be called from Scheme, but of course, other interfaces could be developed by running Scheme code at the interpreter.

This document describes basic usage of **bal** using the built-in functions and gives a full reference manual to the functions that are available from Scheme to aid in writing your own code to automate whatever tasks your financial life needs automated (for example, a script to update the value of your stock or mutual fund holdings by fetching the latest prices from the internet).

2. Why `bal`?

I wrote `bal` because, well, there was not a Free Software solution to manage personal finances on GNU/Linux that had what I wanted in a personal finance program. I wanted a program that was (1) interactive, (2) command-line oriented (for my low RAM laptop), and (3) easy to extend – so I wrote one.

The interactive programs were large GUI programs that were not easy-to-extend. The lightweight programs were non-interactive command line programs that generated reports from plain-text files. So I wrote `bal` to be the interactive, lightweight, and extensible personal finance manager I was looking for. I use it for managing all my personal finances, and I am hopeful that it will be a useful contribution to the Free Software world as well. Let me know if you find it useful or would find any additions useful.

3. Usage

3.1. Creating accounts

3.2. Transferring between accounts

3.3. Browsing by date

3.4. Browsing by pattern

Extending `bal` in Scheme

3.5. Data types

There are two data types in `bal` of interest to programming it in Scheme: the account and the transaction. This brief section discusses the key concepts.

Transaction and Account Numbers. Each transaction has a "coordinate" given by the pair (cons account_number transaction_number). Transactions and accounts are numbered starting at 0, accounts are numbered in the order in which they were created, and transactions are in chronological order by day. Transaction number is the element number in the list returned by `(bal/get-all-transactions account_name)`. Account number is the element number of the account in the list returned by `(bal/get-all-accounts)`.

Accounts in Scheme. When Scheme functions return accounts they do so as a four-element list,

```
(account_name account_type number_of_transactions opening_balance)
```

Transactions in Scheme. When Scheme functions return transactions they do so as a five-element list,

```
(description amount year month day)
```

3.6. Interactive functions

The command prompt is a Scheme interpreter except that the outer-most S expression should not be enclosed in parenthesis to save on typing. This section describes the "interactive functions" — functions that can be entered without arguments at the prompt and then prompt the user for options — available in `bal`.

When entering dollar amounts and other argument options (besides strings), Scheme expressions can be entered. For example, the "amount" of a transaction could be entered as: `(+ 2.99 1.43)`.

3.6.1. Functions to add accounts and transactions

`(aa)`

Add account. Prompts for the name of the account, the kind of account it is, and its opening balance.

`(at)`

Add transaction. Prompts for the account to add the transaction to, the amount of the transaction, a description of the transaction, and when the transaction was made.

`(t)`

Add a transfer between accounts. This is the most commonly-used command because most financial events involve a transaction taking money from one account and another transaction putting money in the other. Prompts for the "from" and "to" accounts, the amount of the transfer, a description of the transaction, and when the transaction took place.

3.6.2. Functions to edit accounts transactions

`(ea)`

Rename account.

`(et [n])`

Edit transaction. Prompts for the transaction to edit and all the various properties of a transaction as if you were adding the transaction anew. Entering nothing for a cateogry leaves it unchanged. Optionally, include a different number of transactions than the default to show when prompting for the transaction (any transaction number can be entered even if it is not displayed in the menu; the menu is just a convenience).

`(da)`

Delete account.

`(dt)`

Delete transaction.

3.6.3. Functions to print account and transaction information

`(la)`

Lists all accounts and their present and future balances.

`(lt)`

Lists the most recent transaction in the current account. The number of transactions to list is set by the Scheme variable `bal/number-to-quick-list` so entering the following at the prompt will change the number to list: `set! bal/number-to-quick-list NUM`

`(ltn)`

The same as `(lt)` except that it prompts for the number to list and for the account to list the transactions from.

`(ltbd)`

list transactions in current account by day, prompting for the first and last day to show.

`(bt)`

List total balances in Expense, Income, Asset, and Liability accounts and total Worth (Asset+Liability) versus Opening Balances for all accounts.

`(re)`

Search for transactions in the current account with a description that matches a regular expression (prompted for).

`(baod)`

Show account balance on different days for the current account. Prompts for a first and last day to show and at what frequency to show the balance in days (for example, specify 7 for weekly).

`(exod)`

Show the total balance across all expense accounts over days.

`(inod)`

Show the total balance across all income accounts over days.

`(liod)`

Show the total balance across all liability accounts over days.

`(wood)`

Show net worth (Assets+Liabilities) over days.

`(ttbd)`

Total all transactions in the current account between a first and last day (split into intervals of a given size).

3.6.4. Utility functions

`(sa)`

Set current account.

`(ca)`

Print current account.

`(w)`

Write account to file (prompted for name).

`(r)`

Read saved accounts from file.

`(p X)`

Print the Scheme object `X`.

`(v)`

Print out the `bal` version.

`(sd)`

Set current day. `(bt)` and `(la)` will now total transactions before and after this day.

`(cd)`

Print current day in YYYY-MM-DD format.

`(q)`

quit `bal`

3.7. Creating interactive functions

To create interactive functions, use the Scheme function

```
(bal/call func options)
```

`func` is a string giving the function name to call, and `options` is a list of pairs containing (in its car) the prompt to give the user and the type of the argument to require (in its cdr). The following types are recognized in `bal`.

string — the option will be treated as a string in the function call

account — the option will be to select an *existing* account (the name of which will be passed as a string to the function call).

current_account — the name of the current account will be passed as a string (the user will not see a prompt for this option).

type — prompt to select an account type (Asset, Liability, Income, Expense).

transaction — prompt to select an existing transaction, passed as a pair giving the account number and the transaction number.

day — prompt to select a year, a month, and a day, passed as a list with three integers in the following order: day, month, year.

other — passed exactly as entered (the user can enter any Scheme expression and it will just be copied as an argument to the function).

Example. The interactive function `(t)` creates a transfer from one account to another account. It is written in the following way,

```
(define t
  (lambda ()
    (bal/call "bal/t"
              (list
               (cons "To Account" "account")
               (cons "From Account" "account")
               (cons "Amount" "real")
               (cons "Description" "string")
               (cons "Day" "day")))))
```

`bal/t` is also a Scheme function. It adds a negative transactions to the "from account" and a positive transaction to the "to account". Its source is,

```
(define bal/t
  (lambda (to-account from-account amount desc day)
    (let ((to-type (list-ref (bal/get-account to-account) 1))
          (from-type (list-ref (bal/get-account from-account) 1)))
      (bal/at to-account amount desc day)
      (bal/at from-account (* -1 amount) desc day))))
```

3.8. Non-interactive functions

`(bal/at account amount desc day)`

adds a transaction to an account where `account` is the name of the account, `amount` is the amount of the transaction, `desc` is a string describing the transaction, and `day` gives the day of the transaction in YYYY-MM-DD format.

`(bal/aa name type ob)`

adds a new account with name *name* and *type* is one of ("expense", "income", "asset", "liability") and gives the type of the account and *ob* gives the opening balance for the account.

`(bal/ea current_account_name new_name)`

rename account from `current_account_name` to `new_name.`

`(bal/da account_name)`

delete account with name `account_name.`

`(bal/dt (cons account_number transaction_number))`

delete transaction.

`(bal/get-current-account)`

returns a string with the name of the current account.

`(bal/get-number-of-accounts)`

return the number of accounts.

`(bal/get-transactions account_name how_many)`

Return

of the latest transactions from account with

`(bal/get-all-transactions account_name)`

Return all transactions from `account_name.` Each transaction is a five element list with elements (description, amount, year, month, day).

`(bal/get-transactions-by-regex account_name regex)`

Return all transactions from `account_name` with descriptions that match `regex.`

`(bal/get-account account_name)`

Returns the account with name `account_name,` a four element list, (name,type_of_account,number_transactions,opening_balance).

`(bal/get-all-accounts)`

Returns a list of all the accounts where each account is a four element list, (name,type_of_account,number_transactions,opening_balance).

`(bal/get-transaction-by-location account_num transact_num)`

Returns the transaction at account number and transaction number, a five-element list (description, amount, year, month, day).

`(bal/get-account-by-location account_num)`

Return account corresponding to `acocunt_num.`

`(bal/get-transactions-by-day account_name first_day last_day)`

Return a list of transactions between the `first_day` and `last_day` within the account with name days`account_name.` Both

`(bal/total-account account_name)`

Returns the sum of all transactions within the account with name, `account_name.`

`(bal/total-all-accounts)`

Returns a list of pairs where each pair has in its `car` the name of the account and in its `cdr` the sum of all transactions within that account.

`(bal/total-by-account-type)`

Returns a list of pairs which have in their `car` the name of the account type (Income, Expense, Asset, Liability), "Worth" (Assets + Liabilities), and "Balances" (for total opening balances) and in its `cdr` the total sum of transactions within each account type.

`(bal/set-account account_name)`

Sets the current account to `account_name.`

`(bal/write file)`

Writes all accounts to `file.`

`(bal/read file)`

Read in accounts from `file.`

`(bal/get-current-file)`

Returns the name of the current default save file.

`(bal/set-select-transact-number num)`

Sets number of transactions to show when selecting a transaction to `num.` On any transaction selection screen you can enter any transaction number whether it is displayed.

`(bal/v)`

Returns a string giving the version of `bal.`

`(bal/t to_account from_account amount desc day)`

Transfers from `from_account` to `to_account` a transaction in `amount` with description `desc` on day (in YYYY-MM-DD format) `day.`

`(bal/set-current-day (list day month year))`

Sets the current day to the year, month, and day provided as arguments. The function returns the list passed to it, unaltered.

`(bal/get-current-day)`

Returns a three element list, `(list day month year),` giving the current day, month, and year.

`(bal/prompt)`

Returns a string which is used as the prompt. Can be redefined to customize the prompt.

`(bal/print-tscts x)`

Print a list of transactions (where transactions are in Scheme format as returned by `(bal/get...transactions...)`).

`(bal/edit-transact loc day amount desc)`

Edit the transaction at location `loc` to be on `day` for `amount` and be described by `desc.` Note that the location of the transaction may change after this command.

`(bal/day-from-time x)`

Return a day object (a list with elements day, month, year) from a Scheme `time-utc` object.

`(bal/seq-days first-day last-day by)`

Return a list of days between first-day and last-day going by `by` days at a time.

`(bal/balance-account-on-days first-day last-day by account)`

Like `baod` but returns the data as a list of pairs with the day in its `car` and the values in its `cdr.`

`(bal/total-transact-in-account-between-days first-day last-day by account)`

Like `ttbd` but allows another account name besides the current account to be specified and it returns its results as a list of pairs with day in the `car` and the values in the `cdr.`

`(bal/output-by-day day amount)`Output formatted output for a gi ven day and an amount. Used to produce output for the on-,over-,between-days class of commands.

`(bal/get-by-type-over-days first-day last-day by num)`

Return a list of pairs with the day in the `car` and the total balance for row `num` (starting at row 0) of `(bt)` output in the `cdr` over the date range given by first-day, last-day, and by.

4. FreeBSD Documentation License