bal - an extensible tool for keeping accounts in order and studying past spending habits

Zach Flynn

ABSTRACT

Gives basic overview of the package and provides a full reference manual to the various Guile Scheme functions bal exposes. This documentation can be freely modified and redistributed under the terms of the GNU Free Documentation License.

23 October 2018

1. Introduction

bal is an 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. 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`.

2.1. 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.

2.2. 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.

2.3. 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).

2.4. 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`.
- `(q)` - quit `bal`

3. 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 string in YYYY-MM-DD format.
- 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))))
```

4. Non-interactive functions