The Joy of Programming of Joy

[The Guide 2 [Programming of Joy] of] i.

ᚹ

Ruurd Wiersma

1st edition

24-09-02

Introduction

This book is meant as a tutorial for the Programming Language Joy, as well as a maintenance guide.

Each chapter is divided into three parts: a Tutorial section, a reference section, called Builtins, and a Theory section, called Maintenance in later chapters.

Readers of this book who are new to Joy and want to learn the language may want to read the Tutorial sections.

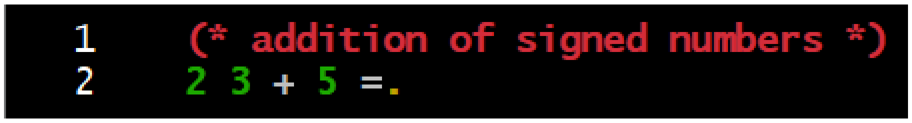
Readers who are experts or do not need the details may want to read the Theory sections. They contain pictures.

Readers who want to implement their own version of Joy can use the reference sections as specification; the tutorial can help with setting up unit tests.

Readers who want to do accounting, can read the chapters A-Z.

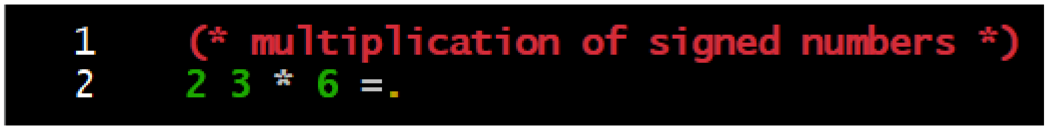
The numbered chapters at the end are meant for those readers who want to read more than what is presented in the earlier chapters.

Tutorial



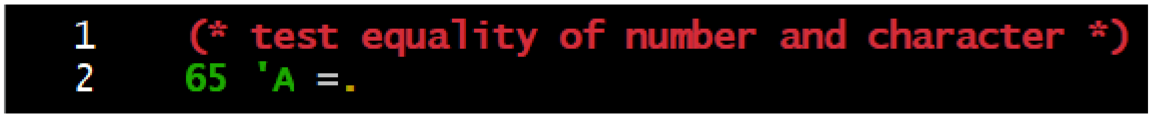
```
(* addition of signed numbers *)
2 3 + 5 =.
```

Addition. More about this in Chapter 1.



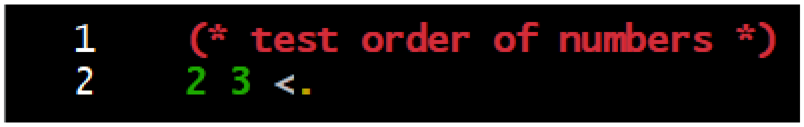
```
(* multiplication of signed numbers *)
2 3 * 6 =.
```

Multiplication. More about this in Chapter 2.



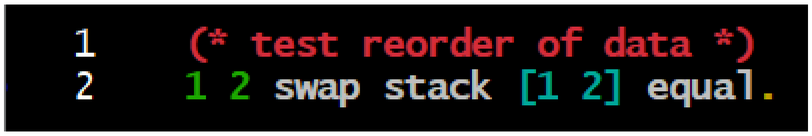
```
(* test equality of number and character *)
65 'A =.
```

Equality test of non-lists. More about this in Chapter 3.



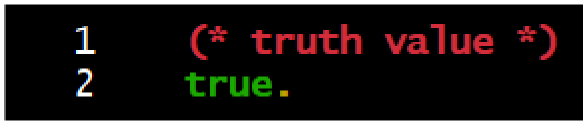
```
(* test order of numbers *)
2 3 <.
```

Comparison. More about this in Chapter 4.



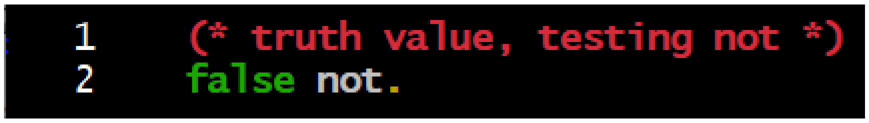
```
(* test reorder of data *)
1 2 swap stack [1 2] equal.
```

Reorder parameters. The words `stack` and `equal` are introduced in a later chapter. More about reordering in Chapter 5.



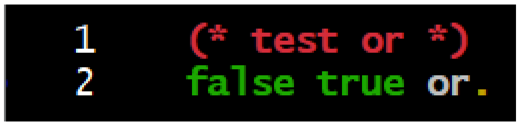
```
(* truth value *)
true.
```

True value. More about this in Chapter 6.



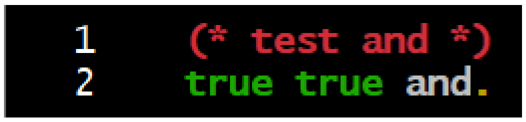
```
(* truth value, testing not *)
false not.
```

False value.



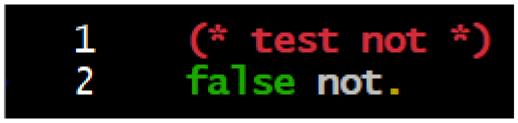
```
(* test or *)
false true or.
```

Or.



```
(* test and *)
true true and.
```

And.



```
(* test not *)
false not.
```

Not.

Builtins

- ⟦+ ...⟧... M I → ⟦...⟧... N
  Numeric N is the result of adding integer I to numeric M.
  Also supports float.
- ⟦* ...⟧... I J → ⟦...⟧... K
  Integer K is the product of integers I and J. Also supports float.
- ⟦= ...⟧... X Y → ⟦...⟧... B
  Either both X and Y are numeric or both are strings or symbols.
  Tests whether X equal to Y. Also supports float.
- ⟦< ...⟧... X Y → ⟦...⟧... B
  Either both X and Y are numeric or both are strings or symbols.
  Tests whether X less than Y. Also supports float.
- ⟦swap ...⟧... X Y → ⟦...⟧... Y X
  Interchanges X and Y on top of the stack.
- ⟦true ...⟧... → ⟦...⟧... true
  Pushes the value true.
- ⟦false ...⟧... → ⟦...⟧... false
  Pushes the value false.
- ⟦or ...⟧... X Y → ⟦...⟧... Z
  Z is the union of sets X and Y, logical disjunction for truth values.
- ⟦and ...⟧... X Y → ⟦...⟧... Z
  Z is the intersection of sets X and Y, logical conjunction for truth values.
- ⟦not ...⟧... X → ⟦...⟧... Y
  Y is the complement of set X, logical negation for truth values.

Theory

Joy is announced as a (purely) functional programming language. What does that mean?

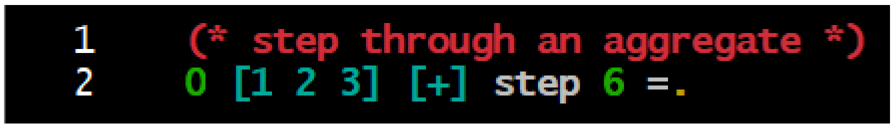
Feature	Supported by Joy
No updates	Yes
No side effects	No
Higher-order functions	Yes
Recursion	Yes
Lazy evaluation	No
Type inference	No
Garbage collection	Yes
Concurrency	No

The first two features are the most important. If supported, the programming language is considered purely functional, whereas it is just functional if only the first feature is present.

The more important question is: what can be done with it? That question will be addressed in the next chapter.

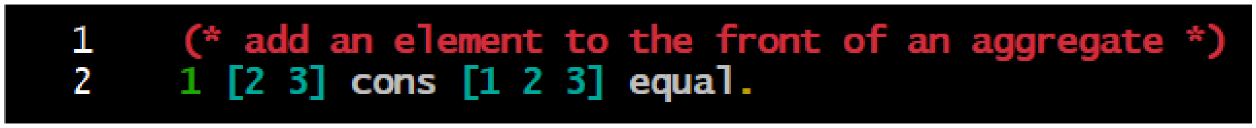
∩

Tutorial



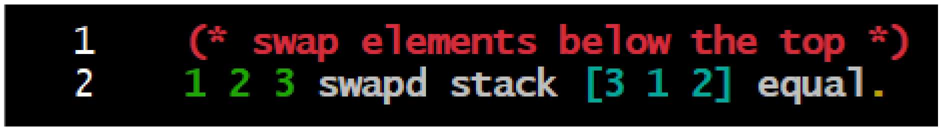
```
(* step through an aggregate *)
0 [1 2 3] [+] step 6 =.
```

Step.



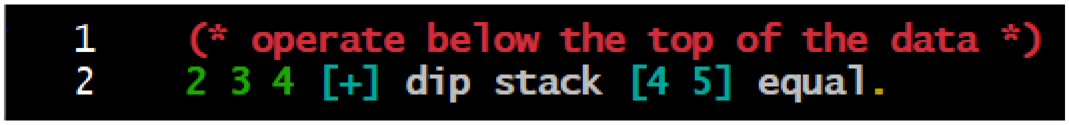
```
(* add an element to the front of an aggregate *)
1 [2 3] cons [1 2 3] equal.
```

Cons.



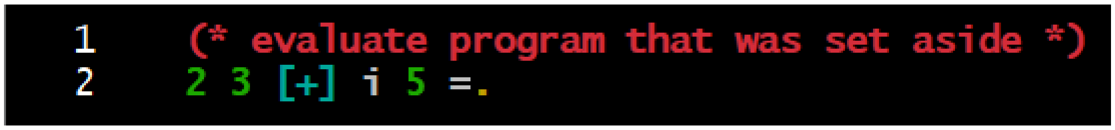
```
(* swap elements below the top *)
1 2 3 swapd stack [3 1 2] equal.
```

Reorder below the top.



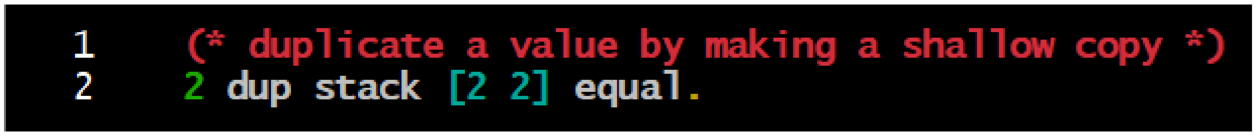
```
(* operate below the top of the data *)
2 3 4 [+] dip stack [4 5] equal.
```

Operate below the top.



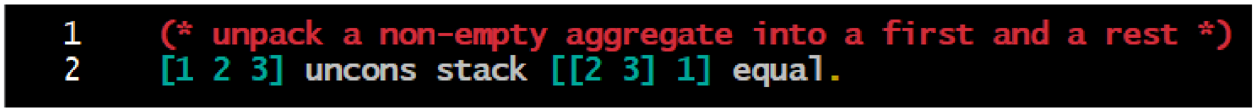
```
(* evaluate program that was set aside *)
2 3 [+] i 5 =.
```

Operate on top.



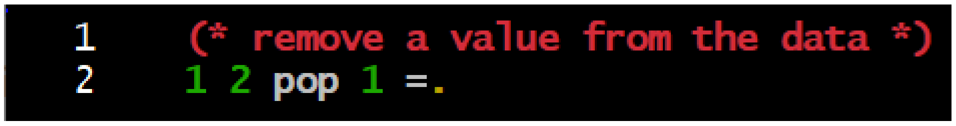
```
(* duplicate a value by making a shallow copy *)
2 dup stack [2 2] equal.
```

Duplicate the top.



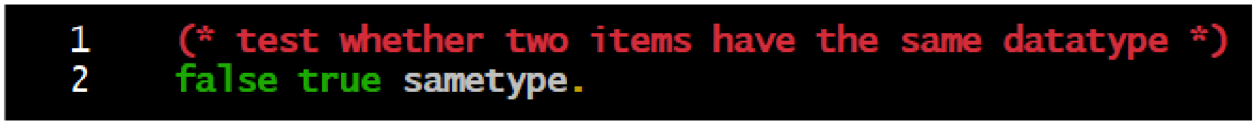
```
(* unpack a non-empty aggregate into a first and a rest *)
[1 2 3] uncons stack [[2 3] 1] equal.
```

Split an aggregate into first and rest.



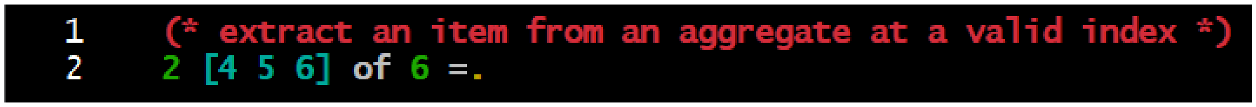
```
(* remove a value from the data *)
1 2 pop 1 =.
```

Remove an item.



```
(* test whether two items have the same datatype *)
false true sametype.
```

Compare types of two items.



```
(* extract an item from an aggregate at a valid index *)
2 [4 5 6] of 6 =.
```

Index an aggregate.

Builtins

- ⟦step ...⟧... A [P] → ⟦A_0 P A_1 P ... A_n P ...⟧...
  Sequentially putting members of aggregate A onto stack,
  executes P for each member of A.
- ⟦cons ...⟧... X A → ⟦...⟧... B
  Aggregate B is A with a new member X (first member for sequences).
- ⟦swapd ...⟧... X Y Z → ⟦...⟧... Y X Z
  As if defined by: swapd == [swap] dip
- ⟦dip ...⟧... X [P] → ⟦P X ...⟧...
  Saves X, executes P, pushes X back.
- ⟦i ...⟧... [P] → ⟦P ...⟧...
  Executes P. So, [P] i == P.
- ⟦dup ...⟧... X → ⟦...⟧... X X
  Pushes an extra copy of X onto stack.
- ⟦uncons ...⟧... A → ⟦...⟧... F R
  F and R are the first and the rest of non-empty aggregate A.
- ⟦pop ...⟧... X → ⟦...⟧...
  Removes X from top of the stack.
- ⟦nothing ...⟧... X → ⟦...⟧... nothing
  [OBSOLETE] Pushes the value nothing.
- ⟦sametype ...⟧... X Y → ⟦...⟧... B
  [EXT] Tests whether X and Y have the same type.
- ⟦of ...⟧... I A → ⟦...⟧... X
  X (= A[I]) is the I-th member of aggregate A.

Theory

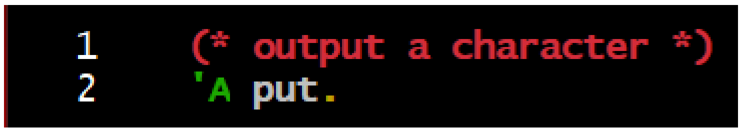
The question is for what purposes the programming language can be used. The following table shows that it can serve a multitude of purposes, just not out-of-the-box:

Application area	Supported by Joy
Calculator	No
Make music	No
Play games	No
Financial accounting	Maybe
Travel planner	No
Online banking	No
DTP	No
Multimedia	No
Internet	No
Artificial intelligence	No

The builtins section contains an obsolete “nothing.” That was used in the definition of the “null” predicate: null == first nothing sametype.

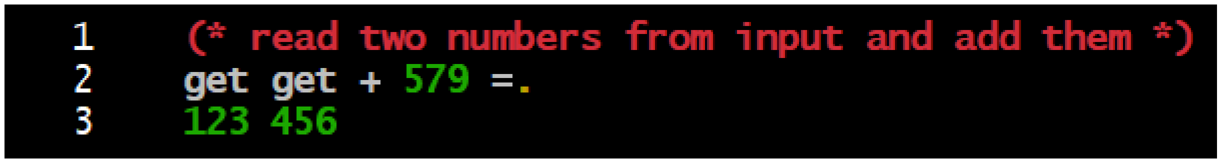
Þ

Tutorial



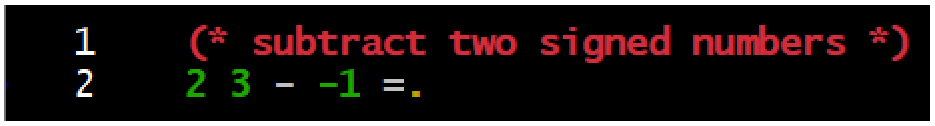
```
(* output a character *)
'A put.
```

Output Joy source code.



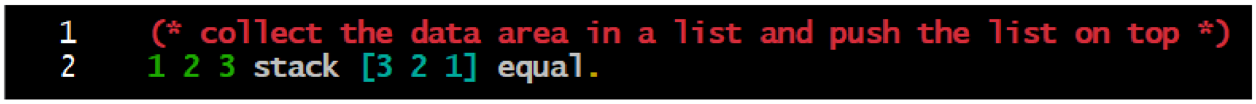
```
(* read two numbers from input and add them *)
get get + 579 =.
123 456
```

Input Joy source code.



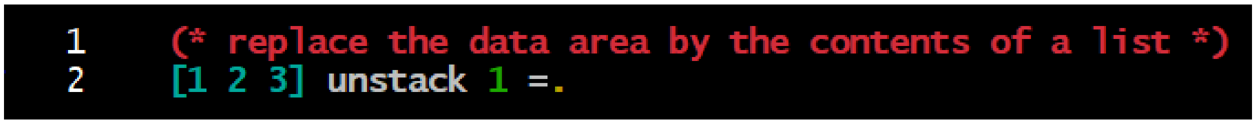
```
(* subtract two signed numbers *)
2 3 - -1 =.
```

Subtract numbers.



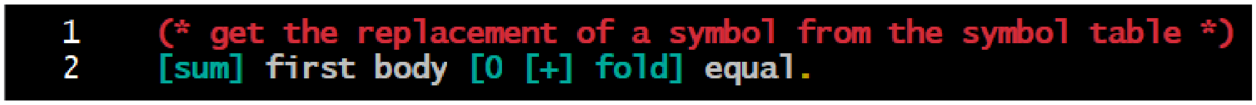
```
(* collect the data area in a list and push the list on top *)
1 2 3 stack [3 2 1] equal.
```

Push a copy of the data area as a list on top.



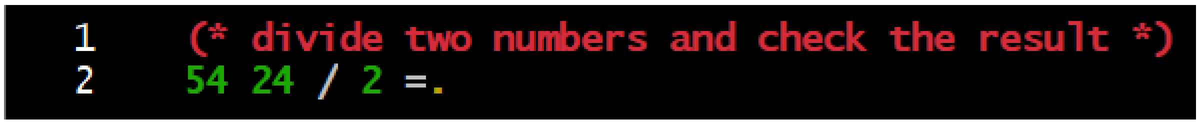
```
(* replace the data area by the contents of a list *)
[1 2 3] unstack 1 =.
```

Replace the contents of the data area with that of a list.



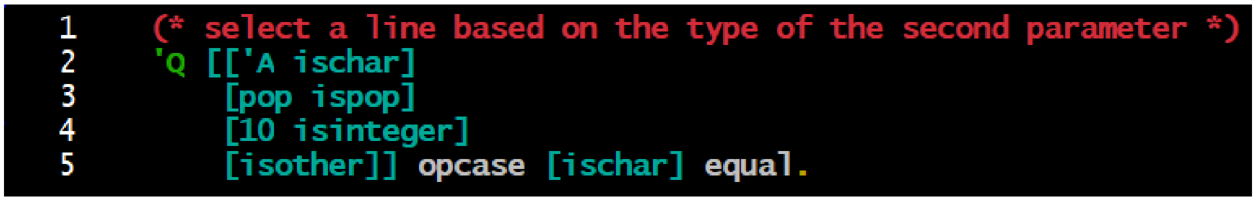
```
(* get the replacement of a symbol from the symbol table *)
[sum] first body [0 [+] fold] equal.
```

Push the body of a user-defined function on top.



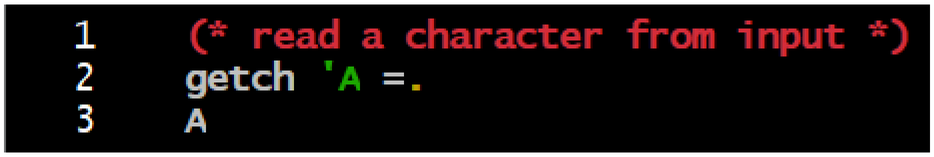
```
(* divide two numbers and check the result *)
54 24 / 2 =.
```

Divide two numbers.



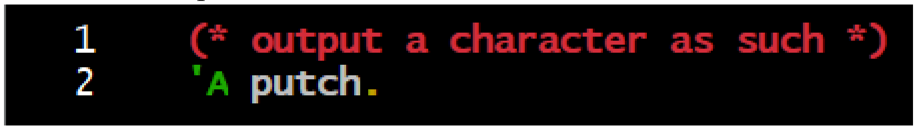
```
(* select a line based on the type of the second parameter *)
'Q [['A ischar]
    [pop ispop]
    [10 isinteger]
    [isother]] opcase [ischar] equal.
```

Opcase.



```
(* read a character from input *)
getch 'A =.
A
```

Read a character from the input file.



```
(* output a character as such *)
'A putch.
```

Write a character to the output file.

Builtins

- ⟦put ...⟧... X → ⟦...⟧...
  [IMPURE] Writes X to output, pops X off stack.
- ⟦get ...⟧... → ⟦...⟧... F
  [IMPURE] Reads a factor from input and pushes it onto stack.
- ⟦- ...⟧... M I → ⟦...⟧... N
  Numeric N is the result of subtracting integer I from numeric M.
  Also supports float.
- ⟦stack ...⟧... X Y Z → ⟦...⟧... X Y Z [Z Y X ...]
  Pushes the stack as a list.
- ⟦unstack ...⟧... [X Y ...] → ⟦...⟧... Y X
  The list [X Y ...] becomes the new stack.
- ⟦body ...⟧... U → ⟦...⟧... [P]
  Quotation [P] is the body of user-defined symbol U.
- ⟦/ ...⟧... I J → ⟦...⟧... K
  Integer K is the (rounded) ratio of integers I and J. Also supports float.
- ⟦opcase ...⟧... X [...[X Xs]...] → ⟦...⟧... X [Xs]
  Indexing on type of X, returns the list [Xs].
- ⟦getch ...⟧... → ⟦...⟧... N
  [IMPURE] Reads a character from input and puts it onto stack.
- ⟦putch ...⟧... N → ⟦...⟧...
  [IMPURE] N : numeric, writes character whose ASCII is N.

Theory

This book is advertized as a maintenance manual. So, what is there to maintain?

Actions	Comments
Remove bugs	There will always be bugs.
Improve data structures	Hash tables, flexible arrays, ...
Add missing features	Local symbols can call each other.
Solve dilemma's	The "Compare" function mitigates a problem in the grammar test file.

Some builtins have been marked as IMPURE and others are marked as FOREIGN. They have side effects, stripping Joy of its purity.

Tutorial

```
(* test order of characters *)
'B 'A >.
```

Greater.

```
(* compare two numbers *)
3 2 >=.
```

Greater or equal.

```
(* compare two characters *)
'A 'B <=.
```

Less or equal.

```
(* verify that integer and character are unequal *)
'A 64 !=.
```

Not equal.

```
(* concatenate two aggregates *)
[1 2 3] [4 5 6] concat [1 2 3 4 5 6] equal.
```

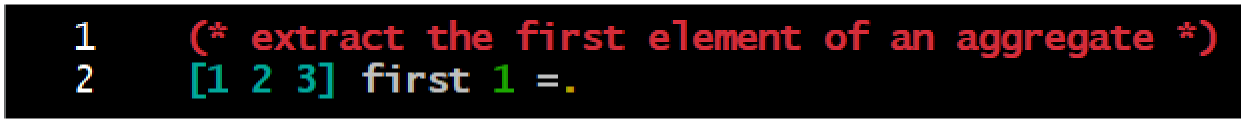
Concatenate.

```
(* check that an empty aggregate is indeed null *)
[] null.
```

Null test.

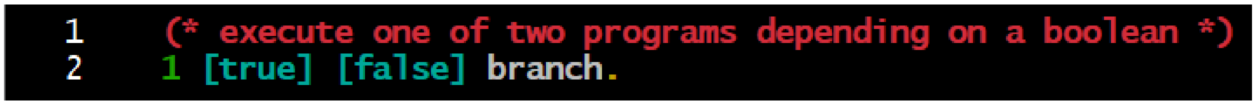
```
(* drop the first member of a non-empty aggregate *)
[1 2 3] rest [2 3] equal.
```

Return everything but the first element.



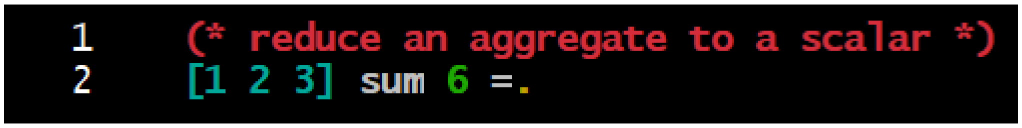
```
(* extract the first element of an aggregate *)
[1 2 3] first 1 =.
```

Extract the first element.



```
(* execute one of two programs depending on a boolean *)
1 [true] [false] branch.
```

Execute one of two programs.



```
(* reduce an aggregate to a scalar *)
[1 2 3] sum 6 =.
```

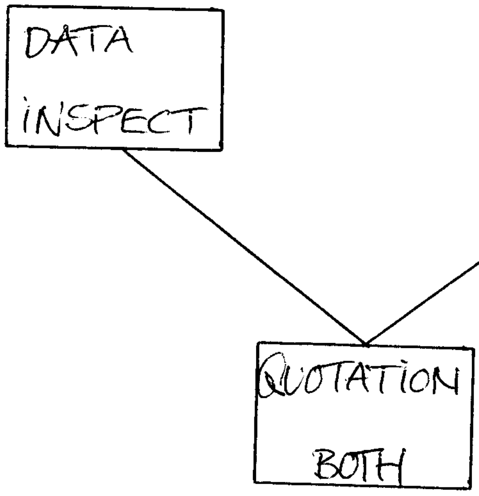
Reduce.

Builtins

- ⟦> ...⟧... X Y → ⟦...⟧... B
  Either both X and Y are numeric or both are strings or symbols.
  Tests whether X greater than Y. Also supports float.
- ⟦>= ...⟧... X Y → ⟦...⟧... B
  Either both X and Y are numeric or both are strings or symbols.
  Tests whether X greater than or equal to Y. Also supports float.
- ⟦<= ...⟧... X Y → ⟦...⟧... B
  Either both X and Y are numeric or both are strings or symbols.
  Tests whether X less than or equal to Y. Also supports float.
- ⟦!= ...⟧... X Y → ⟦...⟧... B
  Either both X and Y are numeric or both are strings or symbols.
  Tests whether X not equal to Y. Also supports float.
- ⟦concat ...⟧... S T → ⟦...⟧... U
  Sequence U is the concatenation of sequences S and T.
- ⟦null ...⟧... X → ⟦...⟧... B
  Tests for empty aggregate X or zero numeric.
- ⟦rest ...⟧... A → ⟦...⟧... R
  R is the non-empty aggregate A with its first member removed.
- ⟦first ...⟧... A → ⟦...⟧... F
  F is the first member of the non-empty aggregate A.
- ⟦branch ...⟧... true [T] [F] → ⟦T...⟧…

  ⟦branch ...⟧... false [T] [F] → ⟦F...⟧...
  If B is true, then executes T else executes F.
- ⟦fold ...⟧... A V_0 [P] → ⟦A_0 P A_1 P ... A_n P ...⟧... V_0
  Starting with value V_0, sequentially pushes members of aggregate A
  and combines with binary operator P to produce value V.

Theory



There have been three design goals in the creation of Joy:

- Program = data;
- Simple;
- Small.

The first one is pictured. A quotation, written as [...] contains a sequence of unevaluated code that descends to the data area where it can be dismembered. The primary ascension operator, or combinator as it is called, is “i”: it takes the contents of a quotation and stacks it on the code area. It should be stated that everything in Joy is a function and as such is both code and data. The code area is used for evaluation; the data area can be used for inspection.

The second one is how Joy got started: what does a programming language look like that does not have variables and also does not have named parameters? And when does the lack of names become inconvenient?

The third one needs an explanation: one-liners are small. More than 100 lines is already becoming large.

simple

The simplicity is also illustrated by the way the inner interpreter operates. It is a fetch-decode-execute cycle where the decoding consists of just two questions:

Is what is decoded a user defined function? If yes, the body of that function is pushed onto the code stack. If not, the second question asks whether it is a builtin. If it is, then the associated C-function is triggered. And if not, whatever was decoded is pushed onto the data area.

small

The small size of it can be illustrated by this live exercise by Manfred:

“I was intrigued by E.W.'s "decorator pattern", e.g. in Joy syntax

```
[Peter Paul Mary] dec => [ [Peter 1] [Paul 2] [Mary 3] ]
```

A trace of my thoughts: "That's just the map combinator ... no, the
numbers have to change ... but map can do that, by leaving the counter
below the list ... oops, no that didn't work ... just use linrec:"

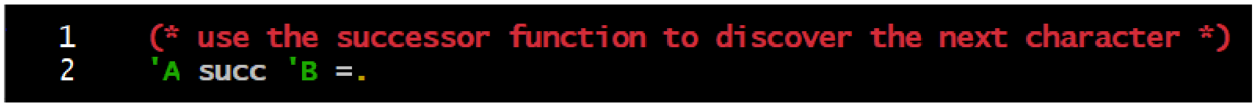
```
DEFINE
dec ==
0 swap
[ null ]
[ popd ]
[ [succ dup] dip uncons swapd ]
[ [swap [] cons cons] dip cons ]
linrec.
```

Not as elegant as I had hoped for, too many dips, swaps, swapds.
I would have preferred to be able to use something like map.
Can anyone think of a better way ?
Or does Joy need a map-like "decorator" combinator ?”

This “dec” is only a function and that is exactly what a Joy program is supposed to do: compute a function.

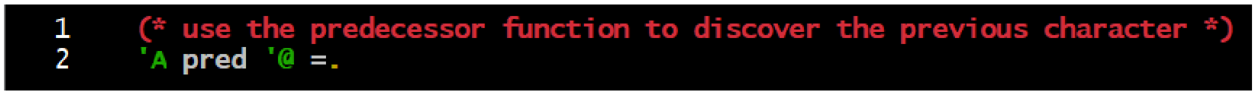
R

Tutorial



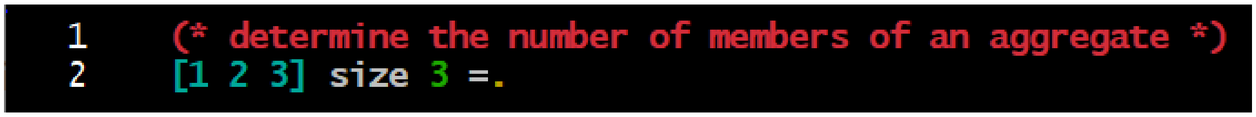
```
(* use the successor function to discover the next character *)
'A succ 'B =.
```

Succ.



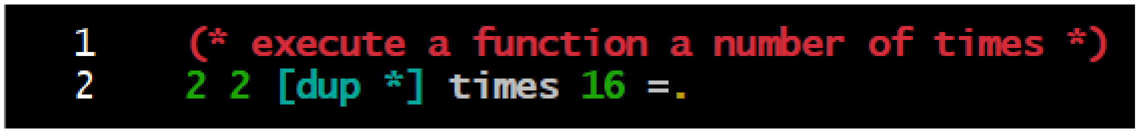
```
(* use the predecessor function to discover the previous character *)
'A pred '@ =.
```

Pred.



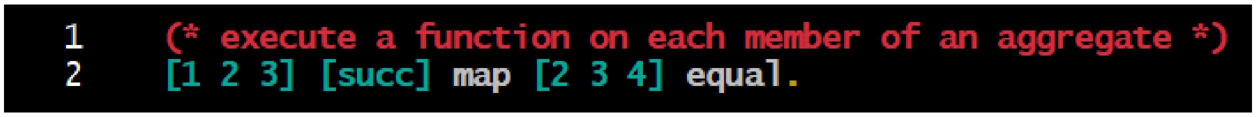
```
(* determine the number of members of an aggregate *)
[1 2 3] size 3 =.
```

Size.



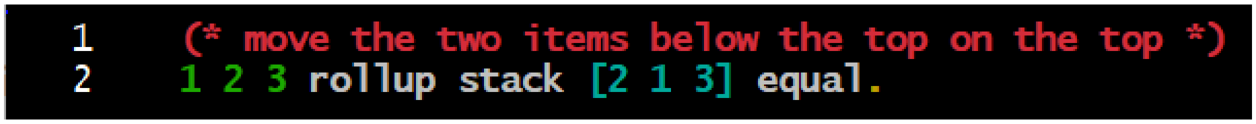
```
(* execute a function a number of times *)
2 2 [dup *] times 16 =.
```

Times.



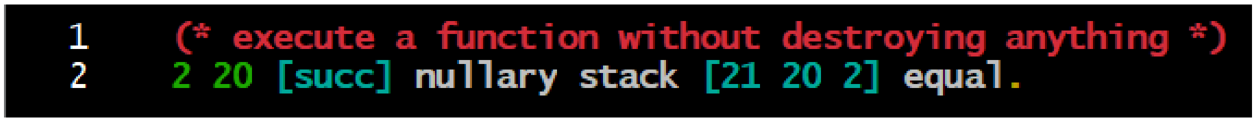
```
(* execute a function on each member of an aggregate *)
[1 2 3] [succ] map [2 3 4] equal.
```

Map.



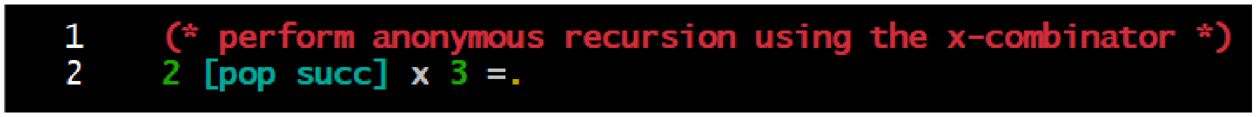
```
(* move the two items below the top on the top *)
1 2 3 rollup stack [2 1 3] equal.
```

Rollup.



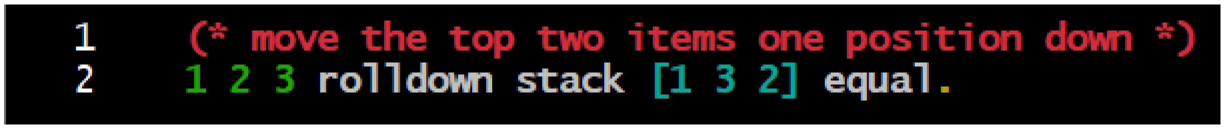
```
(* execute a function without destroying anything *)
2 20 [succ] nullary stack [21 20 2] equal.
```

Nullary.



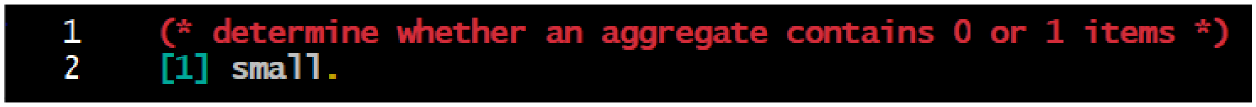
```
(* perform anonymous recursion using the x-combinator *)
2 [pop succ] x 3 =.
```

X.



```
(* move the top two items one position down *)
1 2 3 rolldown stack [1 3 2] equal.
```

Rolldown.



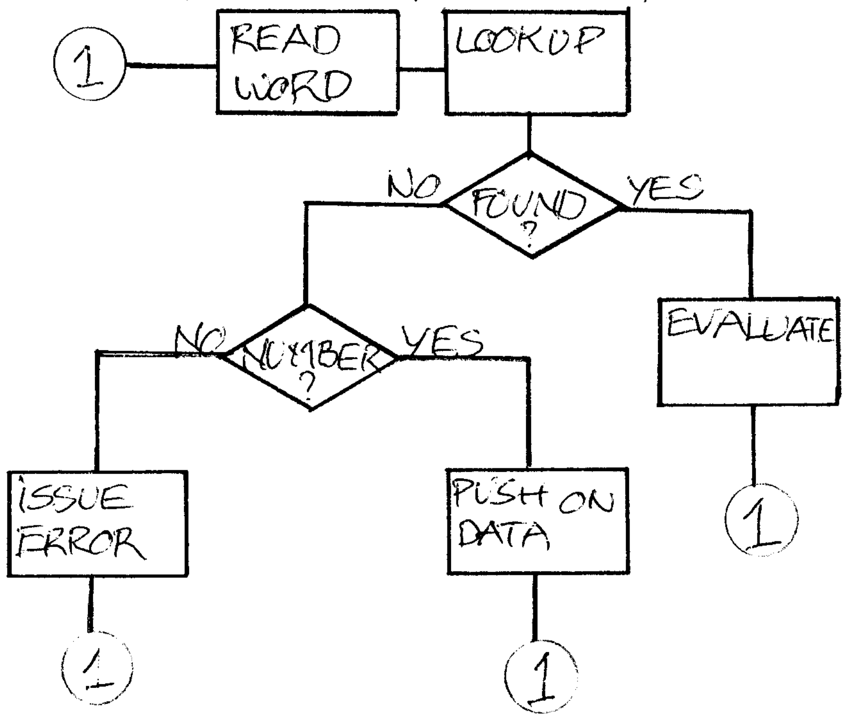
```
(* determine whether an aggregate contains 0 or 1 items *)
[1] small.
```

Small.

Builtins

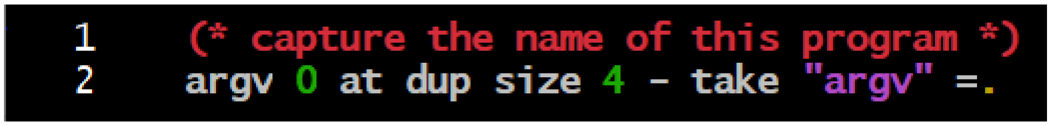
- ⟦succ ...⟧... M → ⟦...⟧... N
  Numeric N is the successor of numeric M.
- ⟦pred ...⟧... M → ⟦...⟧... N
  Numeric N is the predecessor of numeric M.
- ⟦size ...⟧... A → ⟦...⟧... I
  Integer I is the number of elements of aggregate A.
- ⟦times ...⟧... N [P] → ⟦P_1 ... P_n ...⟧...
  N times executes P.
- ⟦map ...⟧... A [P] → ⟦A_0 P ... A_n P ...⟧... B
  Executes P on each member of aggregate A,
  collects results in sametype aggregate B.
- ⟦rollup ...⟧... X Y Z → ⟦...⟧... Z X Y
  Moves X and Y up, moves Z down.
- ⟦nullary ...⟧... [P] → ⟦P ...⟧... R
  Executes P, which leaves R on top of the stack.
  No matter how many parameters this consumes, none are removed from the stack.
- ⟦x ...⟧... [P] → ⟦P ...⟧... [P]
  Executes P without popping [P]. So, [P] x == [P] P.
- ⟦rolldown ...⟧... X Y Z → ⟦...⟧... Y Z X
  Moves Y and Z down, moves X up.
- ⟦small ...⟧... X → ⟦...⟧... B
  Tests whether aggregate X has 0 or 1 members, or numeric 0 or 1.

Theory



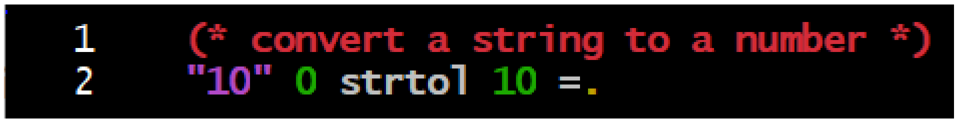
Joy is not the first language without named parameters. Pictured is the outer interpreter of FORTH. Joy differs in that it first reads an entire program and only then starts to evaluate it.

Tutorial



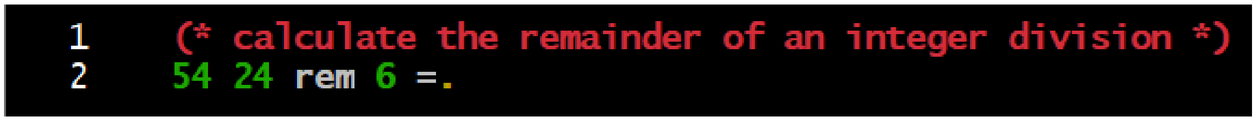
```
(* capture the name of this program *)
argv 0 at dup size 4 - take "argv" =.
```

Argv.



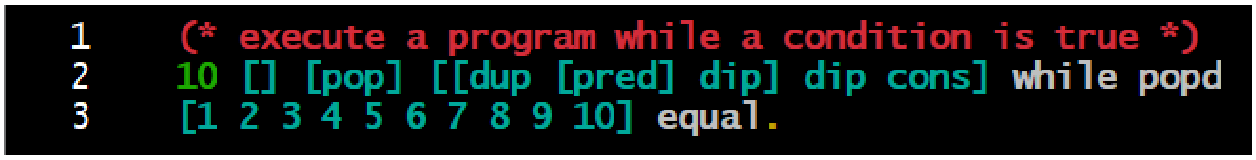
```
(* convert a string to a number *)
"10" 0 strtol 10 =.
```

Strtol.



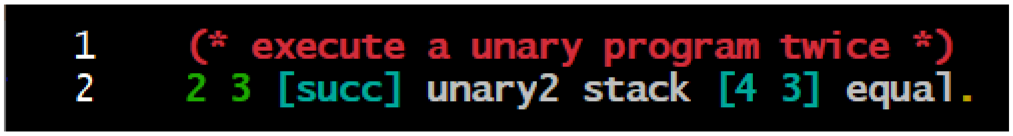
```
(* calculate the remainder of an integer division *)
54 24 rem 6 =.
```

Rem.



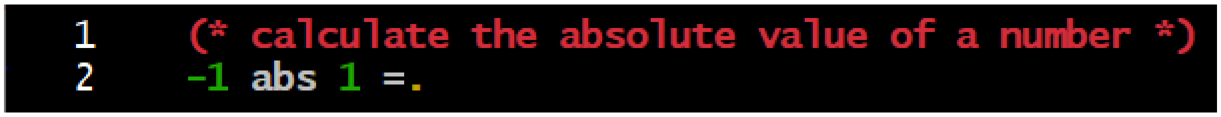
```
(* execute a program while a condition is true *)
10 [] [pop] [[dup [pred] dip] dip cons] while popd
[1 2 3 4 5 6 7 8 9 10] equal.
```

While.



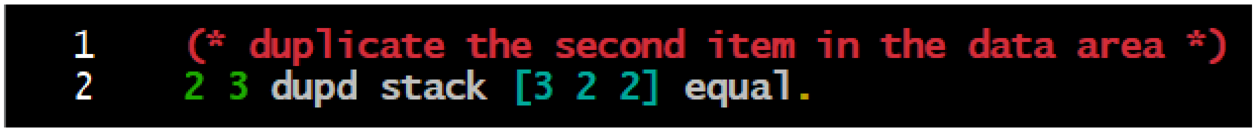
```
(* execute a unary program twice *)
2 3 [succ] unary2 stack [4 3] equal.
```

Unary2.



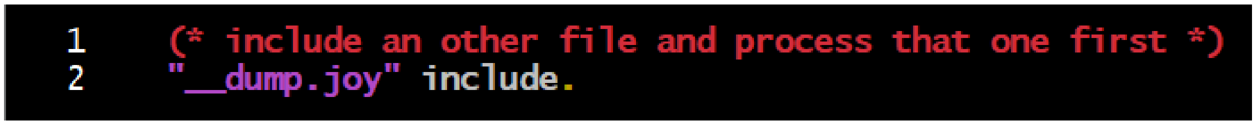
```
(* calculate the absolute value of a number *)
-1 abs 1 =.
```

Abs.



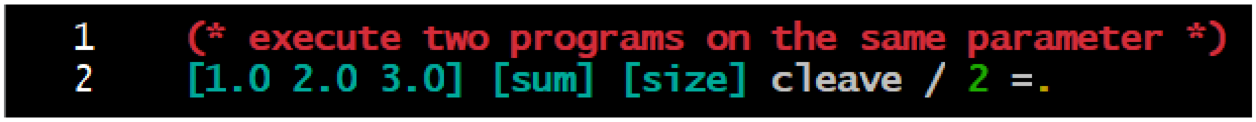
```
(* duplicate the second item in the data area *)
2 3 dupd stack [3 2 2] equal.
```

Dupd.



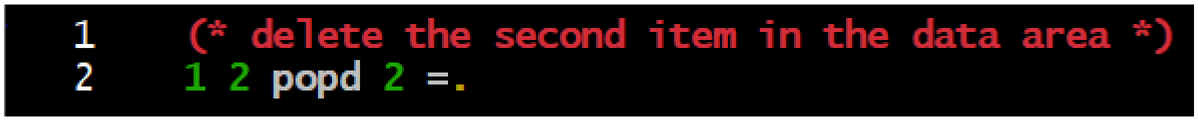
```
(* include an other file and process that one first *)
"__dump.joy" include.
```

Include.



```
(* execute two programs on the same parameter *)
[1.0 2.0 3.0] [sum] [size] cleave / 2 =.
```

Cleave.



```
(* delete the second item in the data area *)
1 2 popd 2 =.
```

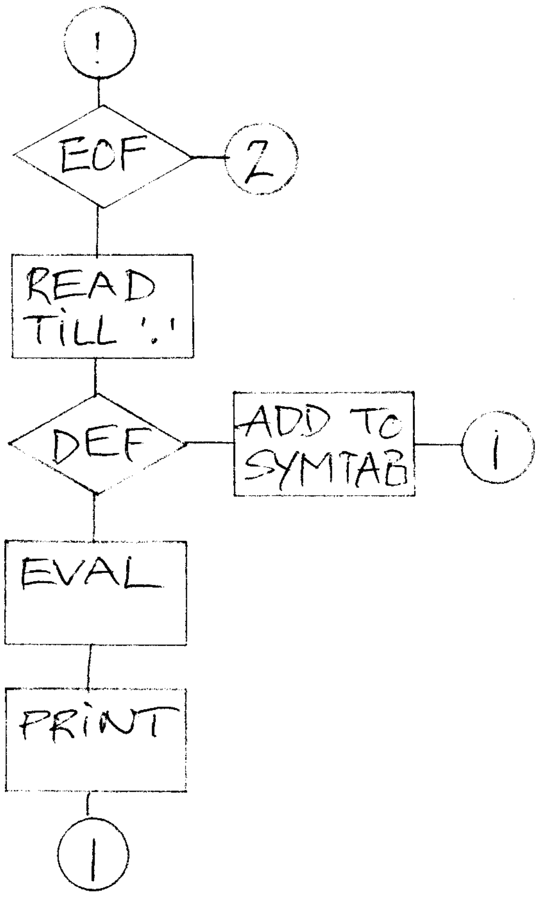
Popd.

Builtins

- ⟦argv ...⟧... → ⟦...⟧... A
  Creates an aggregate A containing the interpreter's command line arguments.
- ⟦strtol ...⟧... S I → ⟦...⟧... J
  String S is converted to the integer J using base I.
  If I = 0, assumes base 10,
  but leading "0" means base 8 and leading "0x" means base 16.
- ⟦rem ...⟧... I J → ⟦...⟧... K
  Integer K is the remainder of dividing I by J. Also supports float.
- ⟦while ...⟧... → ⟦B jump-on-false-to-(1) D [B] [D] while (1) ...⟧...
  While executing B yields true executes D.
- ⟦unary2 ...⟧... X1 X2 [P] → ⟦X1 P move-result-R1 X2 P move-result-R2 ...⟧... R1 R2
  Executes P twice, with X1 and X2 on top of the stack.
  Returns the two values R1 and R2.
- ⟦abs ...⟧... N1 → ⟦...⟧... N2
  Integer N2 is the absolute value (0,1,2..) of integer N1,
  or float N2 is the absolute value (0.0 ..) of float N1.
- ⟦dupd ...⟧... Y Z → ⟦...⟧... Y Y Z
  As if defined by: dupd == [dup] dip
- ⟦include ...⟧... "filnam.ext" → ⟦...⟧...
  Transfers input to file whose name is "filnam.ext".
  On end-of-file returns to previous input file.
- ⟦cleave ...⟧... X [P1] [P2] → ⟦X P1 move-result-R1 X R2 move-result-R2 ...⟧... R1 R2
  Executes P1 and P2, each with X on top, producing two results.
- ⟦popd ...⟧... Y Z → ⟦...⟧... Z
  As if defined by: popd == [pop] dip

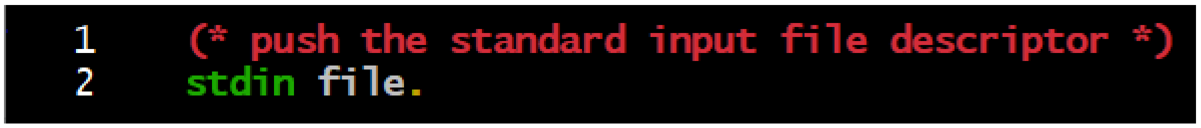
Theory

Pictured is the outer interpreter of Joy. It reads definitions and programs. Definitions are stored in the symbol table; programs are evaluated and their top result is printed. The outer interpreter is finished at end of file.



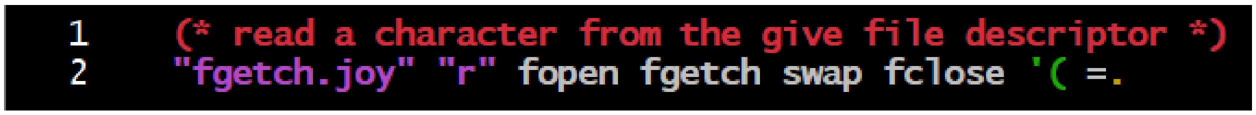
X

Tutorial



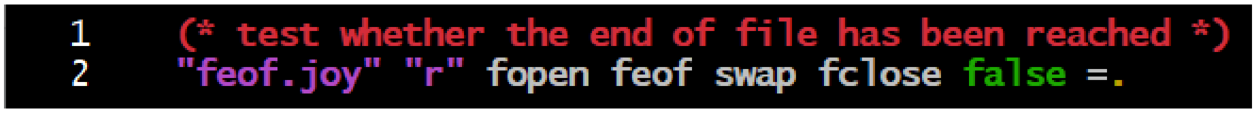
```
(* push the standard input file descriptor *)
stdin file.
```

Stdin.



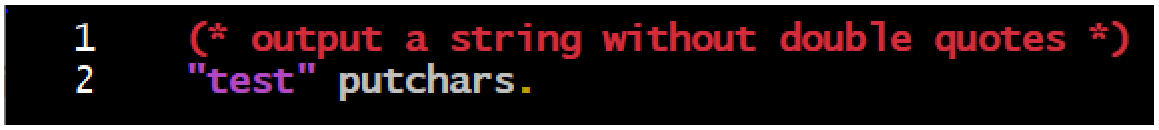
```
(* read a character from the give file descriptor *)
"fgetch.joy" "r" fopen fgetch swap fclose '( =.
```

Fgetch.



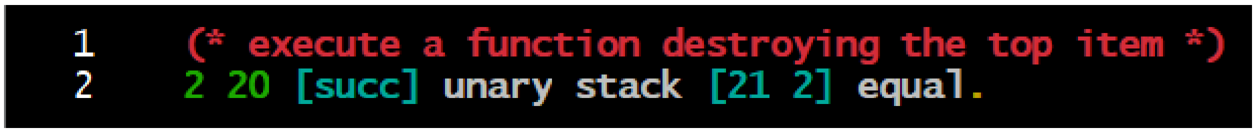
```
(* test whether the end of file has been reached *)
"feof.joy" "r" fopen feof swap fclose false =.
```

Feof.



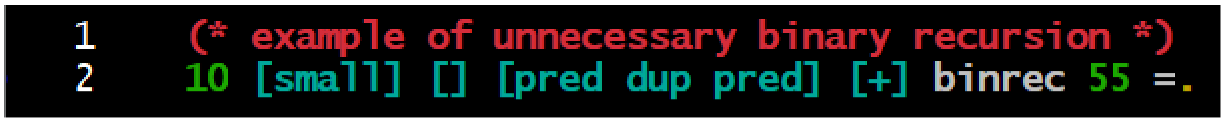
```
(* output a string without double quotes *)
"test" putchars.
```

Putchars.



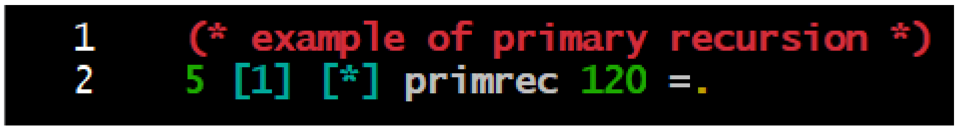
```
(* execute a function destroying the top item *)
2 20 [succ] unary stack [21 2] equal.
```

Unary.



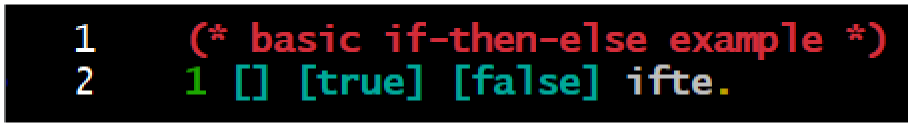
```
(* example of unnecessary binary recursion *)
10 [small] [] [pred dup pred] [+] binrec 55 =.
```

Binrec.



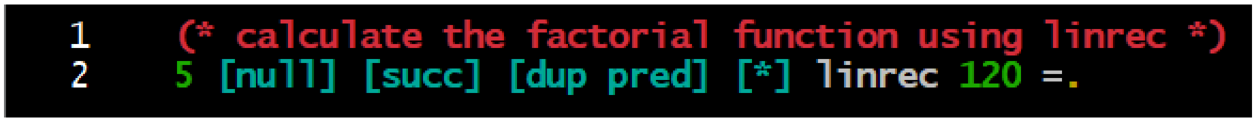
```
(* example of primary recursion *)
5 [1] [*] primrec 120 =.
```

Primrec.



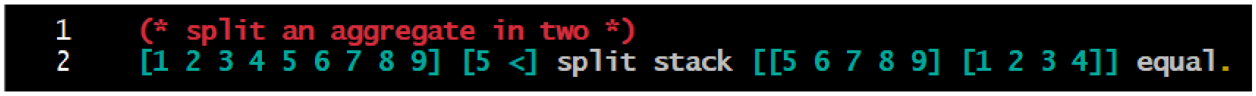
```
(* basic if-then-else example *)
1 [] [true] [false] ifte.
```

Ifte.



```
(* calculate the factorial function using linrec *)
5 [null] [succ] [dup pred] [*] linrec 120 =.
```

Linrec.



```
(* split an aggregate in two *)
[1 2 3 4 5 6 7 8 9] [5 <] split stack [[5 6 7 8 9] [1 2 3 4]] equal.
```

Split.

Builtins

- ⟦stdin ...⟧... → ⟦...⟧... S
  [FOREIGN] Pushes the standard input stream.

- ⟦fgetch ...⟧... S → ⟦...⟧... S C
  [FOREIGN] C is the next available character from stream S.

- ⟦feof ...⟧... S → ⟦...⟧... S B
  [FOREIGN] B is the end-of-file status of stream S.

- ⟦putchars ...⟧... "abc..." → ⟦...⟧...
  [IMPURE] Writes abc.. (without quotes)

- ⟦unary ...⟧... X [P] → ⟦P ...⟧… R
  Executes P, which leaves R on top of the stack.
  No matter how many parameters this consumes,
  exactly one is removed from the stack.

- ⟦binrec ...⟧... [P] [T] [R1] [R2] → ⟦P jump-on-false-to-(1) T jump-to-(2) (1) R1 [P] [T] [R1] [R2] binrec swap [P] [T] [R1] [R2] binrec R2 (2) ...⟧… R
  Executes P. If that yields true, executes T.
  Else uses R1 to produce two intermediates, recurses on both,
  then executes R2 to combine their results.

- ⟦primrec ...⟧... X [I] [C] → ⟦...⟧… R
  Executes I to obtain an initial value R0.
  For integer X uses increasing positive integers to X, combines by C for new R.
  For aggregate X uses successive members and combines by C for new R.

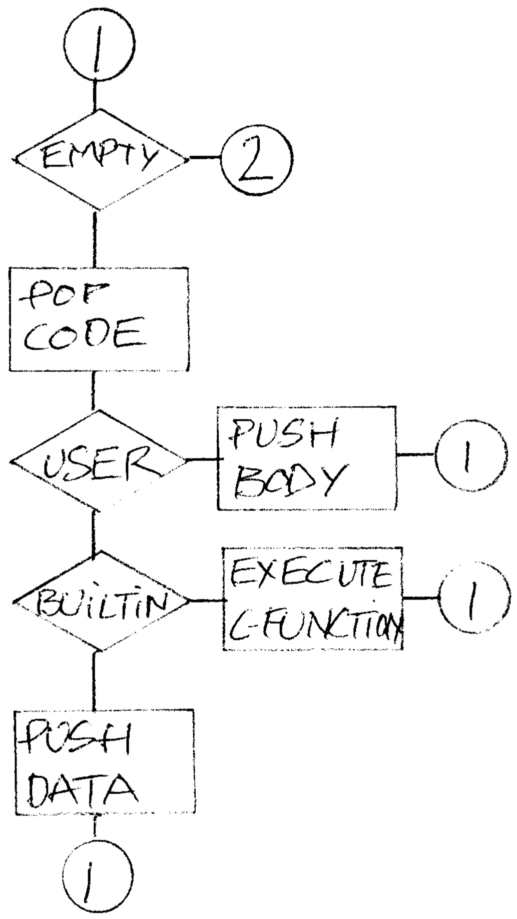
- ⟦ifte ...⟧... [B] [T] [F] → ⟦B jump-on-false-to-(1) T jump-to-(2) (1) F (2) ...⟧…
  Executes B. If that yields true, then executes T else executes F.

- ⟦linrec ...⟧... [P] [T] [R1] [R2] → ⟦P jump-on-false-to-(1) T jump-to-(2) (1) R1 [P] [T] [R1] [R2] linrec R2 (2) ...⟧…
  Executes P. If that yields true, executes T.
  Else executes R1, recurses, executes R2.

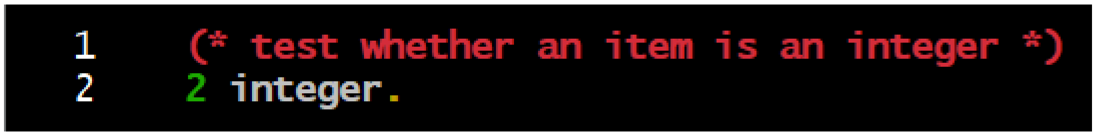
- ⟦split ...⟧... A [B] → ⟦A_0 B ... A_n B ...⟧... A1 A2
  Uses test B to split aggregate A into sametype aggregates A1 and A2.

Theory

Pictured is the inner interpreter of Joy. It takes a value from the code stack. If it is a user defined function, the body is pushed onto the code stack; if it is a builtin the associated C function is executed and in all other cases the value is pushed onto the data stack. The inner interpreter is finished when the code stack becomes empty.

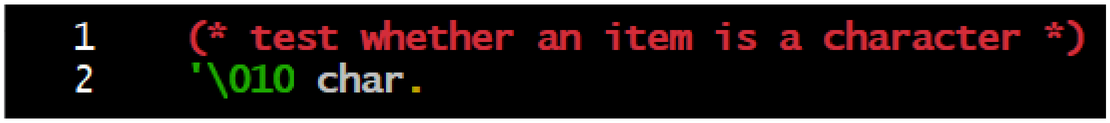


Tutorial



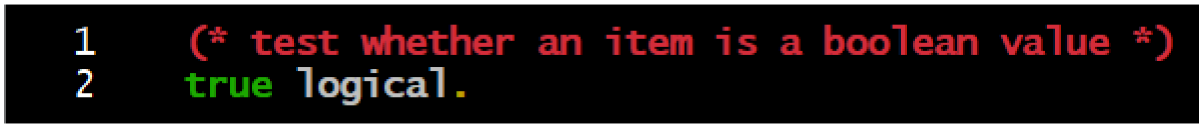
```
(* test whether an item is an integer *)
2 integer.
```

Integer.



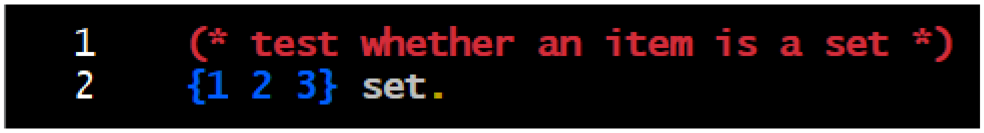
```
(* test whether an item is a character *)
'\010 char.
```

Char.



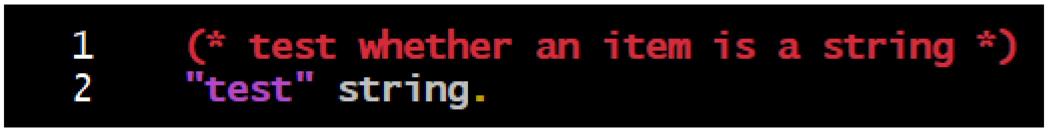
```
(* test whether an item is a boolean value *)
true logical.
```

Logical.



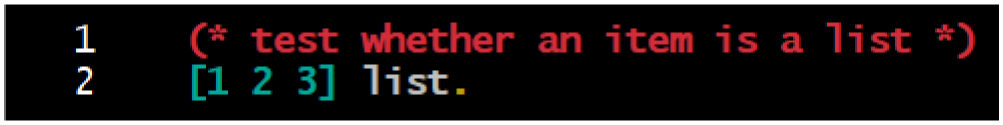
```
(* test whether an item is a set *)
{1 2 3} set.
```

Set.



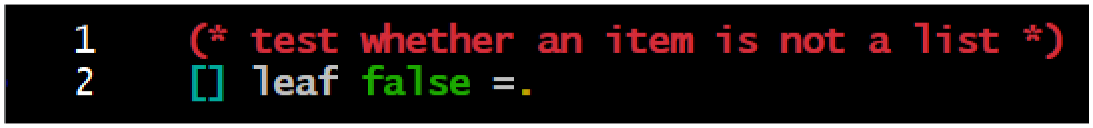
```
(* test whether an item is a string *)
"test" string.
```

String.



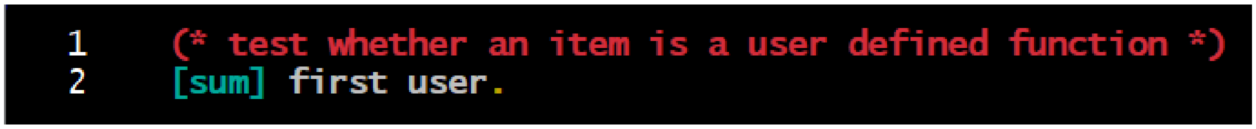
```
(* test whether an item is a list *)
[1 2 3] list.
```

List.



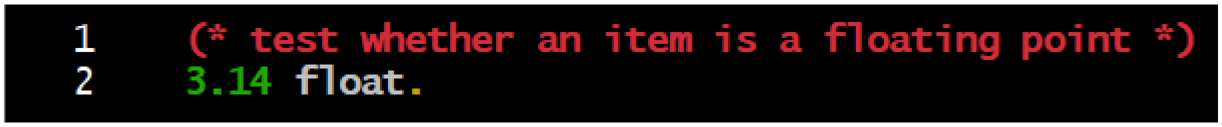
```
(* test whether an item is not a list *)
[] leaf false =.
```

Leaf.



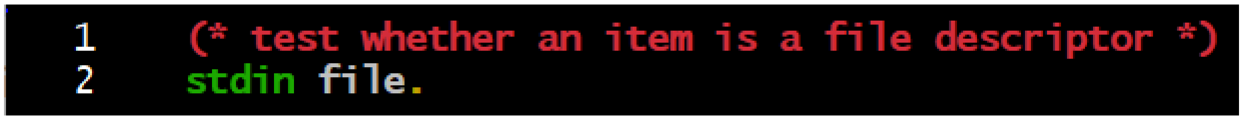
```
(* test whether an item is a user defined function *)
[sum] first user.
```

User.



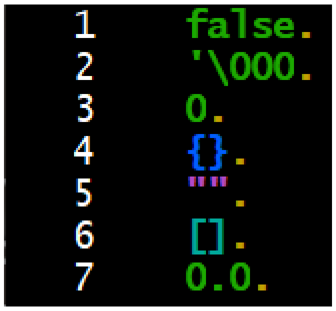
```
(* test whether an item is a floating point *)
3.14 float.
```

Float.



```
(* test whether an item is a file descriptor *)
stdin file.
```

File.



```
false.
'\000.
0.
{}.
"".
[].
0.0.
```

Each of the datatypes mentioned in the Builtins section has its own way of expressing nothing. All nothings compare equal, even though each one of them has a different datatype. There are two exceptions: user defined symbols cannot be 0 and builtin functions also cannot be 0. Files can be 0 when there is a failure to open a file. There is no constant that can express this kind of 0.

Builtins

- ⟦integer ...⟧... X → ⟦...⟧... B
  Tests whether X is an integer.
- ⟦char ...⟧... X → ⟦...⟧... B
  Tests whether X is a character.
- ⟦logical ...⟧... X → ⟦...⟧... B
  Tests whether X is a logical.
- ⟦set ...⟧... X → ⟦...⟧... B
  Tests whether X is a set.
- ⟦string ...⟧... X → ⟦...⟧... B
  Tests whether X is a string.
- ⟦list ...⟧... X → ⟦...⟧... B
  Tests whether X is a list.
- ⟦leaf ...⟧... X → ⟦...⟧... B
  Tests whether X is not a list.
- ⟦user ...⟧... X → ⟦...⟧... B
  Tests whether X is a user-defined symbol.
- ⟦float ...⟧... X → ⟦...⟧... B
  Tests whether R is a float.
- ⟦file ...⟧... X → ⟦...⟧... B
  [FOREIGN] Tests whether F is a file.

Theory

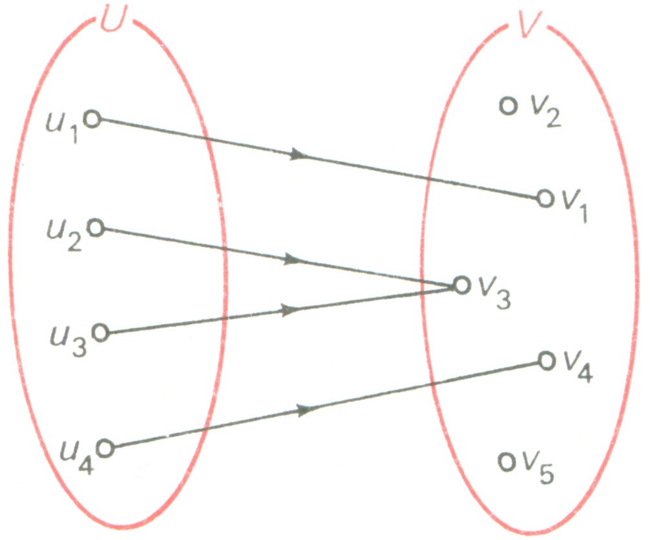
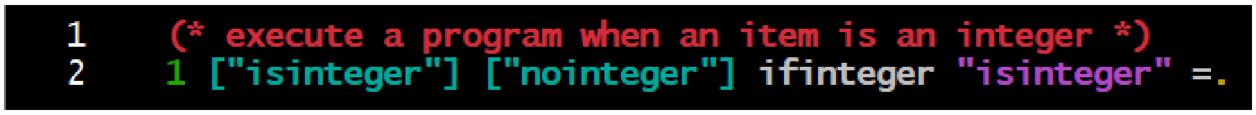


FIG. 2.1

Joy is a functional programming language. What is a function? This picture, taken from a book about mathematics shows a relation between U and V. Both are sets. A function is a special kind of relation: all elements of the domain are connected to exactly one element of the codomain. So, there is a function U → V. There is no function from V to U.

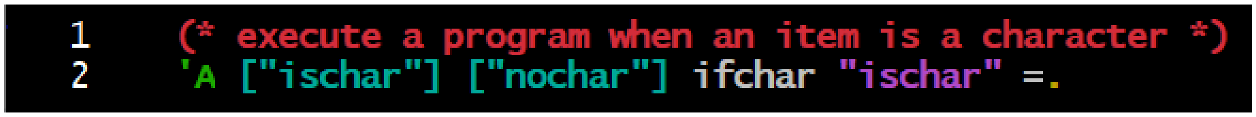
H

Tutorial



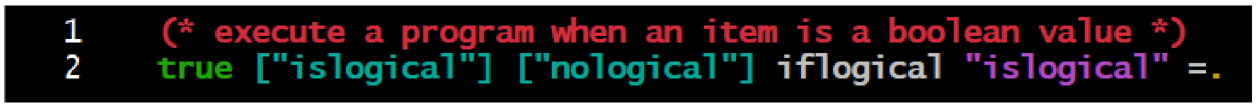
```
(* execute a program when an item is an integer *)
1 ["isinteger"] ["nointeger"] ifinteger "isinteger" =.
```

Ifinteger.



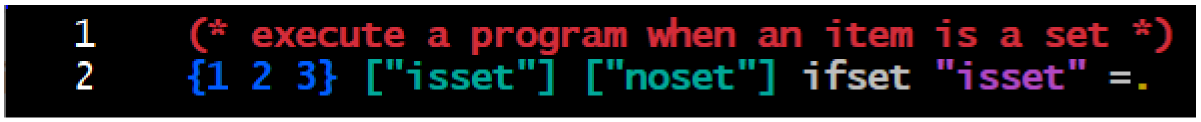
```
(* execute a program when an item is a character *)
'A ["ischar"] ["nochar"] ifchar "ischar" =.
```

Ifchar.



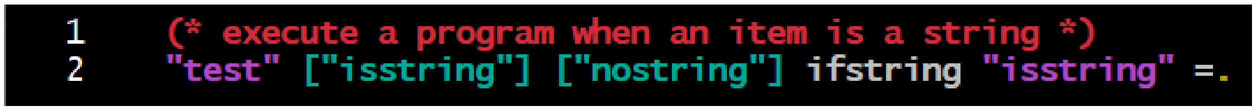
```
(* execute a program when an item is a boolean value *)
true ["islogical"] ["nological"] iflogical "islogical" =.
```

Iflogical.



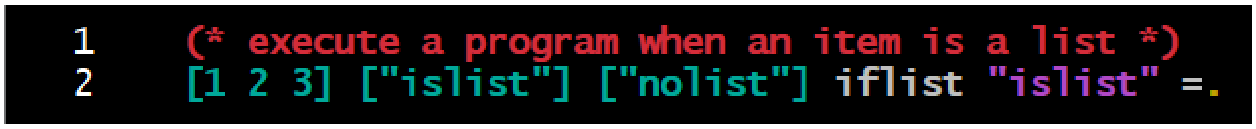
```
(* execute a program when an item is a set *)
{1 2 3} ["isset"] ["noset"] ifset "isset" =.
```

Ifset.



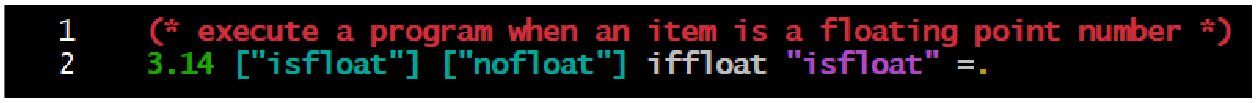
```
(* execute a program when an item is a string *)
"test" ["isstring"] ["nostring"] ifstring "isstring" =.
```

Ifstring.



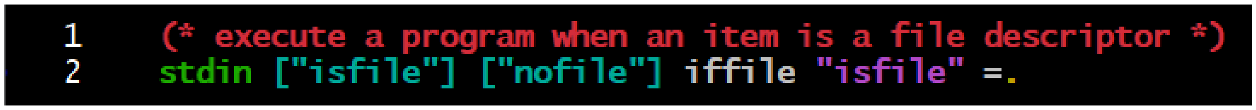
```
(* execute a program when an item is a list *)
[1 2 3] ["islist"] ["nolist"] iflist "islist" =.
```

Iflist.



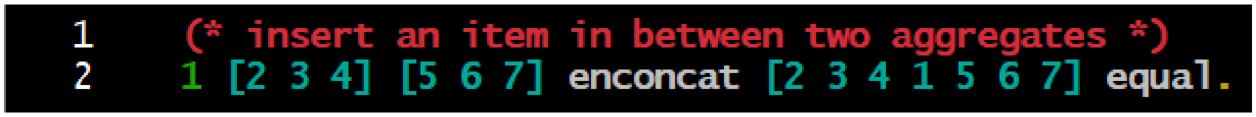
```
(* execute a program when an item is a floating point number *)
3.14 ["isfloat"] ["nofloat"] iffloat "isfloat" =.
```

Iffloat.



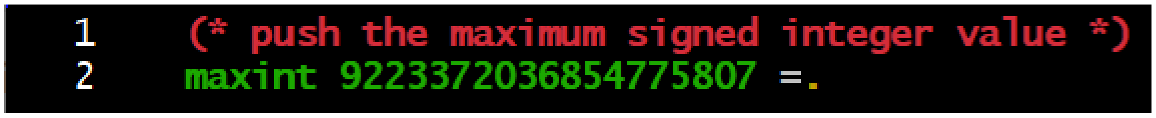
```
(* execute a program when an item is a file descriptor *)
stdin ["isfile"] ["nofile"] iffile "isfile" =.
```

Iffile.



```
(* insert an item in between two aggregates *)
1 [2 3 4] [5 6 7] enconcat [2 3 4 1 5 6 7] equal.
```

Enconcat.



```
(* push the maximum signed integer value *)
maxint 9223372036854775807 =.
```

Maxint.

Builtins

- ⟦ifinteger ...⟧... X [T] [E] → ⟦T/F ...⟧...
  If X is an integer, executes T else executes E.
- ⟦ifchar ...⟧... X [T] [E] → ⟦T/F ...⟧...
  If X is a character, executes T else executes E.
- ⟦iflogical ...⟧... X [T] [E] → ⟦T/F ...⟧...
  If X is a logical or truth value, executes T else executes E.
- ⟦ifset ...⟧... X [T] [E] → ⟦T/F ...⟧...
  If X is a set, executes T else executes E.
- ⟦ifstring ...⟧... X [T] [E] → ⟦T/F ...⟧...
  If X is a string, executes T else executes E.
- ⟦iflist ...⟧... X [T] [E] → ⟦T/F ...⟧...
  If X is a list, executes T else executes E.
- ⟦iffloat ...⟧... X [T] [E] → ⟦T/F ...⟧...
  If X is a float, executes T else executes E.
- ⟦iffile ...⟧... X [T] [E] → ⟦T/F ...⟧...
  [FOREIGN] If X is a file, executes T else executes E.
- ⟦enconcat ...⟧... X S T → ⟦...⟧... U
  Sequence U is the concatenation of sequences S and T
  with X inserted between S and T (== swapd cons concat).
- ⟦maxint ...⟧... → ⟦...⟧... maxint
  Pushes largest integer (platform dependent). Typically it is 32 bits.

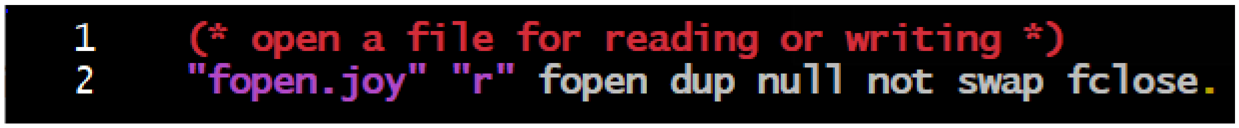
Theory

Continuing from the previous chapter, this is another way to present a function:

Domain	Codomain
1	1
2	2
3	3
4	5
5	8
6	13
7	21
8	34
9	55
10	89
11	144
12	233
13	377

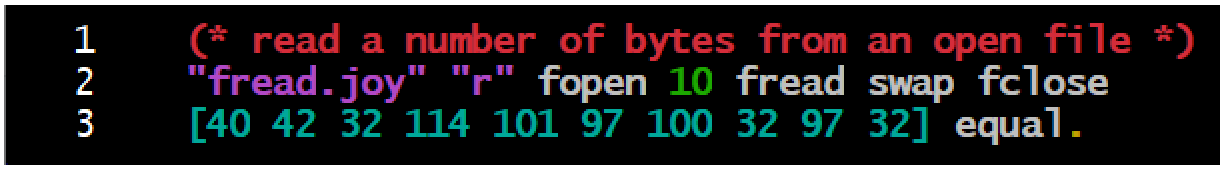
As can be seen from these examples, there is no notion of computation. The usual formulation: “a function must return the same value, when given the same input” comes from physics or computer science, not mathematics.

Tutorial



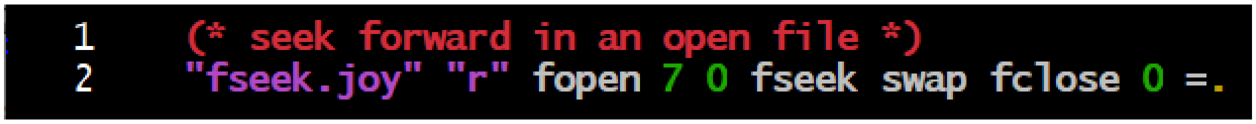
```
(* open a file for reading or writing *)
"fopen.joy" "r" fopen dup null not swap fclose.
```

Fopen.



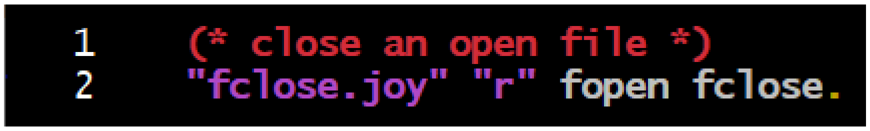
```
(* read a number of bytes from an open file *)
"fread.joy" "r" fopen 10 fread swap fclose
[40 42 32 114 101 97 100 32 97 32] equal.
```

Fread.



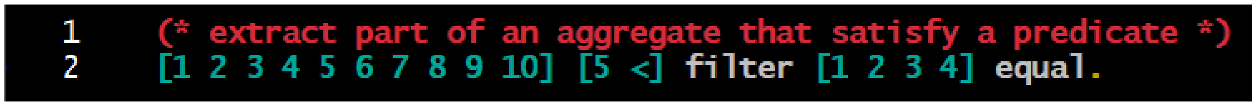
```
(* seek forward in an open file *)
"fseek.joy" "r" fopen 7 0 fseek swap fclose 0 =.
```

Fseek.



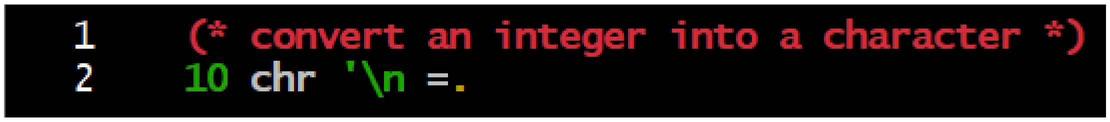
```
(* close an open file *)
"fclose.joy" "r" fopen fclose.
```

Fclose.



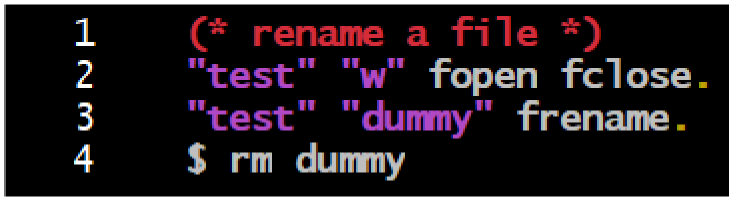
```
(* extract part of an aggregate that satisfy a predicate *)
[1 2 3 4 5 6 7 8 9 10] [5 <] filter [1 2 3 4] equal.
```

Filter.



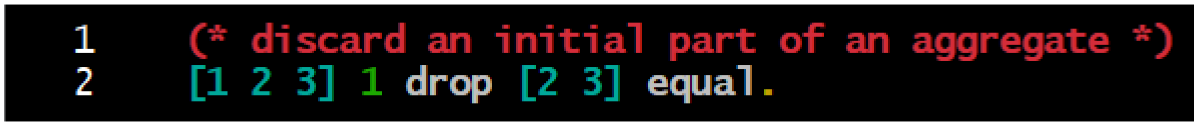
```
(* convert an integer into a character *)
10 chr '\n =.
```

Chr.



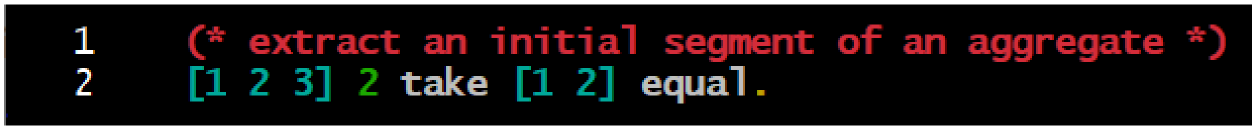
```
(* rename a file *)
"test" "w" fopen fclose.
"test" "dummy" frename.
$ rm dummy
```

Frename.



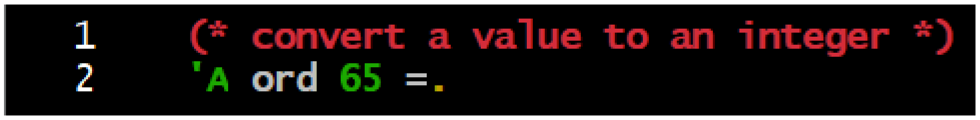
```
(* discard an initial part of an aggregate *)
[1 2 3] 1 drop [2 3] equal.
```

Drop.



```
(* extract an initial segment of an aggregate *)
[1 2 3] 2 take [1 2] equal.
```

Take.



```
(* convert a value to an integer *)
'A ord 65 =.
```

Ord.

Builtins

- ⟦fopen ...⟧... P M → ⟦...⟧... S
  [FOREIGN] The file system object with pathname P is opened with mode M (r, w, a, etc.) and stream object S is pushed; if the open fails, file:NULL is pushed.
- ⟦fread ...⟧... S I → ⟦...⟧... S L
  [FOREIGN] I bytes are read from the current position of stream S and returned as a list of I integers.
- ⟦fseek ...⟧... S P W → ⟦...⟧... S B
  [FOREIGN] Stream S is repositioned to position P relative to whence-point W, where W = 0, 1, 2 for beginning, current position, end respectively.
- ⟦fclose ...⟧... S → ⟦...⟧...
  [FOREIGN] Stream S is closed and removed from the stack.
- ⟦filter ...⟧... A [B] → ⟦A_0 B ... A_n B ...⟧... A_1
  Uses test B to filter aggregate A producing sametype aggregate A_1.
- ⟦chr ...⟧... I → ⟦...⟧... C
  C is the character whose Ascii value is integer I (or logical or character).
- ⟦frename ...⟧... P1 P2 → ⟦...⟧... B
  [FOREIGN] The file system object with pathname P1 is renamed to P2. B is a boolean indicating success or failure.
- ⟦drop ...⟧... A N → ⟦...⟧... B
  Aggregate B is the result of deleting the first N elements of A.
- ⟦take ...⟧... A N → ⟦...⟧... B
  Aggregate B is the result of retaining just the first N elements of A.
- ⟦ord ...⟧... C → ⟦...⟧... I
  Integer I is the Ascii value of character C (or logical or integer).

Theory

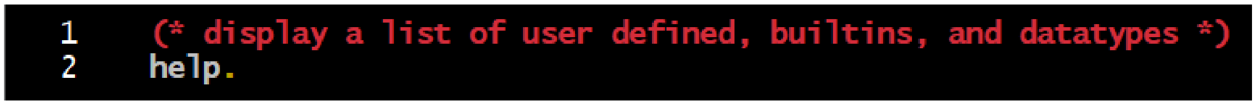
FOREIGN and IMPURE functions have side effects. A function has the property that it can be evaluated anywhere anytime and always gives the same answer when given the same input. They can even be evaluated at compile time. As it happens there are more functions that should not be evaluated at compile time:

Builtin	Runtime	Foreign	Impure
__html_manual	No	No	Yes
__latex_manual	No	No	Yes
__settracegc	No	No	Yes
_help	No	No	Yes
abort	Yes	No	No
argc	Yes	No	No
argv	Yes	No	No
clock	Yes	No	Yes
fclose	Yes	Yes	No
feof	Yes	Yes	No
ferror	Yes	Yes	No
fflush	Yes	Yes	No
fgetch	Yes	Yes	No
fgets	Yes	Yes	No
file	Yes	Yes	No
filetime	Yes	Yes	No
fopen	Yes	Yes	No
fput	Yes	Yes	No
fputch	Yes	Yes	No
fputchars	Yes	Yes	No
fputstring	Yes	Yes	No
fread	Yes	Yes	No
fremove	Yes	Yes	No
frename	Yes	Yes	No
fseek	Yes	Yes	No
ftell	Yes	Yes	No
fwrite	Yes	Yes	No
gc	Yes	No	Yes
get	Yes	No	Yes
getenv	Yes	No	No
help	No	No	Yes
helpdetail	No	No	Yes
manual	No	No	Yes
iffile	Yes	Yes	No
put	Yes	No	Yes

putch	Yes	No	Yes
putchars	Yes	No	Yes
quit	Yes	No	No
rand	Yes	No	Yes
setautoput	No	No	Yes
setecho	No	No	Yes
setundeferror	No	No	Yes
srand	Yes	No	Yes
stderr	Yes	Yes	No
stdin	Yes	Yes	No
stdout	Yes	Yes	No
system	Yes	No	Yes
time	Yes	No	Yes

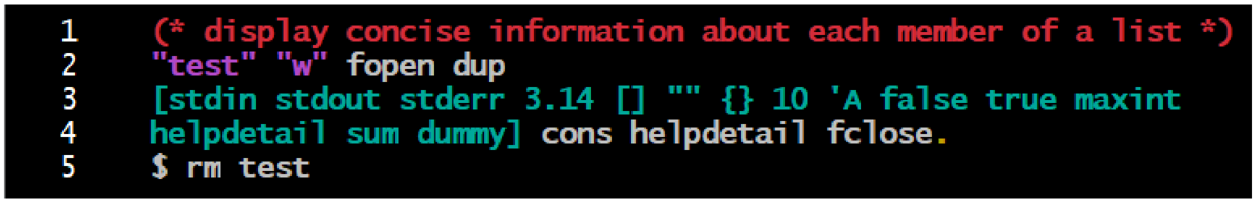
The column Runtime contains the functions that should be deferred till runtime; if No the compiler should ignore the function alltogether. The columns FOREIGN and IMPURE are mutually exclusive. FOREIGN functions are also IMPURE. Only one of the two labels is given to a function.

Tutorial



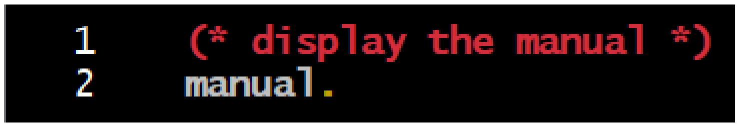
```
(* display a list of user defined, builtins, and datatypes *)
help.
```

Help.



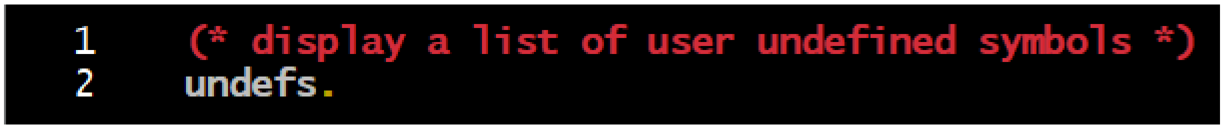
```
(* display concise information about each member of a list *)
"test" "w" fopen dup
[stdin stdout stderr 3.14 [] "" {} 10 'A false true maxint
helpdetail sum dummy] cons helpdetail fclose.
$ rm test
```

Helpdetail.



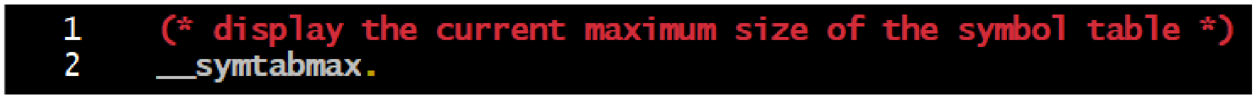
```
(* display the manual *)
manual.
```

Manual.



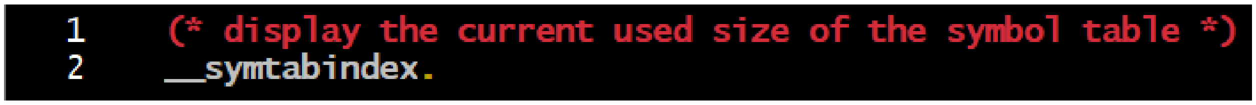
```
(* display a list of user undefined symbols *)
undefs.
```

Undefs.



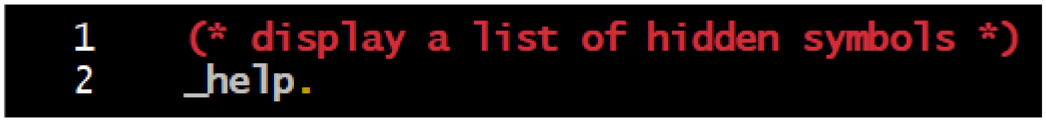
```
(* display the current maximum size of the symbol table *)
__symtabmax.
```

Symtabmax.



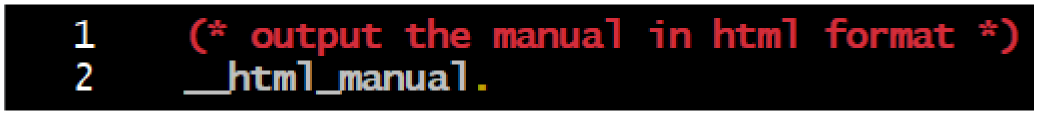
```
(* display the current used size of the symbol table *)
__symtabindex.
```

Symtabindex.



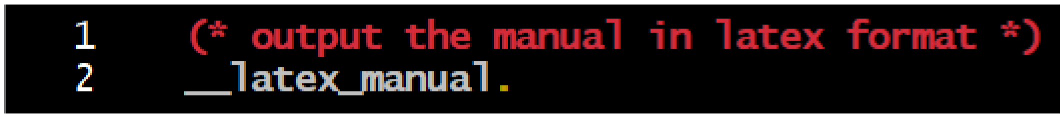
```
(* display a list of hidden symbols *)
_help.
```

_Help.



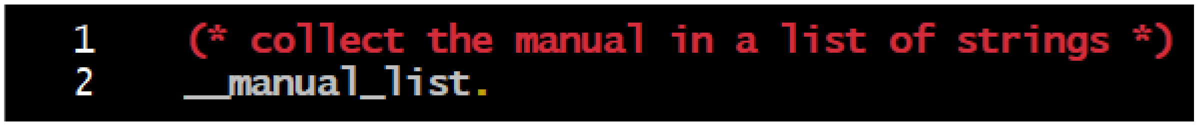
```
(* output the manual in html format *)
__html_manual.
```

Html_manual.



```
(* output the manual in latex format *)
__latex_manual.
```

Latex_manual.



```
(* collect the manual in a list of strings *)
__manual_list.
```

Manual_list.

Builtins

- ⟦help ...⟧... → ⟦...⟧...
  [IMPURE] Lists all defined symbols, including those from library files.
  Then lists all primitives of raw Joy.
  (There is a variant: "_help" which lists hidden symbols).

- ⟦helpdetail ...⟧... [S1 S2 …] → ⟦...⟧...
  [IMPURE] Gives brief help on each symbol S in the list.

- ⟦manual ...⟧... → ⟦...⟧...
  [IMPURE] Writes this manual of all Joy primitives to output file.

- ⟦undefs ...⟧... → ⟦...⟧...

  Push a list of all undefined symbols in the current symbol table.

- ⟦__symtabmax ...⟧... → ⟦...⟧... I
  Pushes value of maximum size of the symbol table.

- ⟦__symtabindex ...⟧... → ⟦...⟧... I
  Pushes current size of the symbol table.

- ⟦_help ...⟧... → ⟦...⟧... I
  [IMPURE] Lists all hidden symbols in library and then all hidden builtin symbols.

- ⟦__html_manual ...⟧... → ⟦...⟧... I
  [IMPURE] Writes this manual of all Joy primitives to output file in HTML style.

- ⟦__latex_manual ...⟧... → ⟦...⟧... I
  [IMPURE] Writes this manual of all Joy primitives to output file in Latex style but without the head and tail.

- ⟦__manual_list ...⟧... → ⟦...⟧... I
  Pushes a list L of lists (one per operator) of three documentation strings.

Theory

Computer Architecture
Tempfile
Kilo.joy
Joy
C
Shell
OS
Hardware

Tutorial

```
(* calculate the acos function *)
0.1 acos 'g 0 6 formatf strtod 1.47063 =.
```

Acos.

```
(* calculate the asin function *)
0.1 asin 'g 0 6 formatf strtod 0.100167 =.
```

Asin.

```
(* calculate the atan function *)
0.1 atan 'g 0 6 formatf strtod 0.0996687 =.
```

Atan.

```
(* calculate the ceil function *)
1.5 ceil 2 =.
```

Ceil.

```
(* calculate the cos function *)
0.5 cos 'g 0 6 formatf strtod 0.877583 =.
```

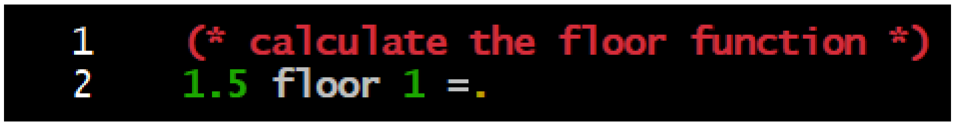
Cos.

```
(* calculate the cosh function *)
0.5 cosh 'g 0 6 formatf strtod 1.12763 =.
```

Cosh.

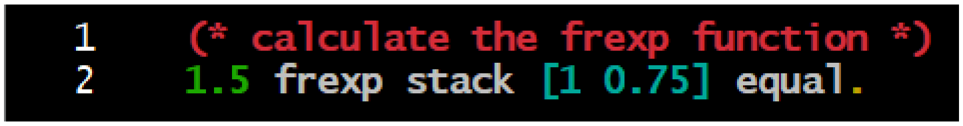
```
(* calculate the exp function *)
1.5 exp 'g 0 6 formatf strtod 4.48169 =.
```

Exp.



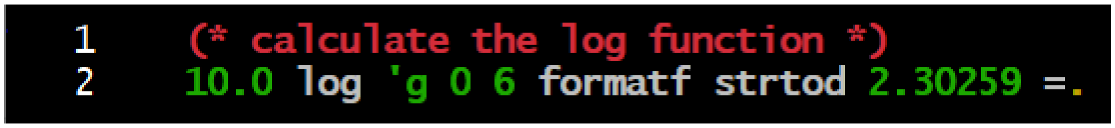
```
(* calculate the floor function *)
1.5 floor 1 =.
```

Floor.



```
(* calculate the frexp function *)
1.5 frexp stack [1 0.75] equal.
```

Frexp.



```
(* calculate the log function *)
10.0 log 'g 0 6 formatf strtod 2.30259 =.
```

Log.

Builtins

- ⟦acos ...⟧... F → ⟦...⟧... G
  G is the arc cosine of F.
- ⟦asin ...⟧... F → ⟦...⟧... G
  G is the arc sine of F.
- ⟦atan ...⟧... F → ⟦...⟧... G
  G is the arc tangent of F.
- ⟦ceil ...⟧... F → ⟦...⟧... G
  G is the float ceiling of F.
- ⟦cos ...⟧... F → ⟦...⟧... G
  G is the cosine of F.
- ⟦cosh ...⟧... F → ⟦...⟧... G
  G is the hyperbolic cosine of F.
- ⟦exp ...⟧... F → ⟦...⟧... G
  G is e (2.718281828...) raised to the Fth power.
- ⟦floor ...⟧... F → ⟦...⟧... G
  G is the floor of F.
- ⟦frexp ...⟧... F → ⟦...⟧... G I
  G is the mantissa and I is the exponent of F.
  Unless F = 0, 0.5 <= abs(G) < 1.0.
- ⟦log ...⟧... F → ⟦...⟧... G
  G is the natural logarithm of F.

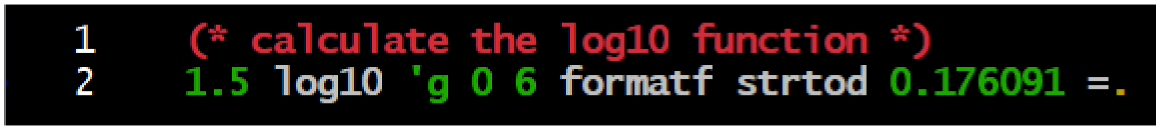
Theory

Program	Lines
OS	12000
Compiler	4000
Editor	1000
Joy	182

The OS is MINIX, the compiler is the P4 Pascal compiler, the editor is KILO, and the Joy program is lsplib.joy. Lines are including comments, so there is some room to manoeuvre in case an extra feature needs to be added.

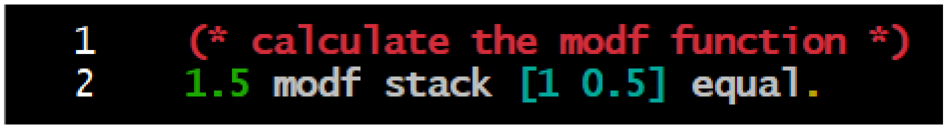
√

Tutorial



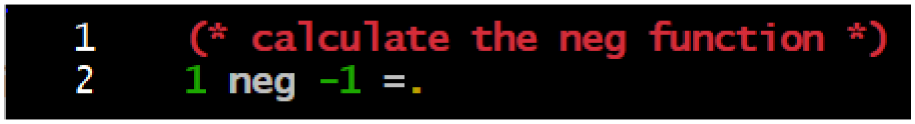
```
(* calculate the log10 function *)
1.5 log10 'g 0 6 formatf strtod 0.176091 =.
```

Log10.



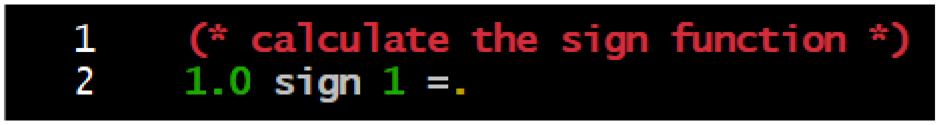
```
(* calculate the modf function *)
1.5 modf stack [1 0.5] equal.
```

Modf.



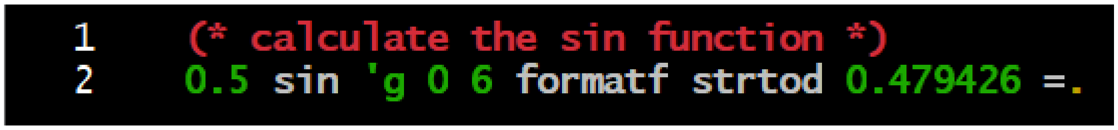
```
(* calculate the neg function *)
1 neg -1 =.
```

Neg.



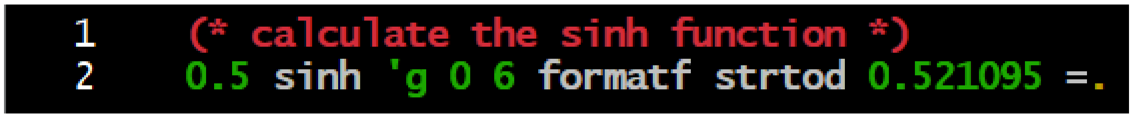
```
(* calculate the sign function *)
1.0 sign 1 =.
```

Sign.



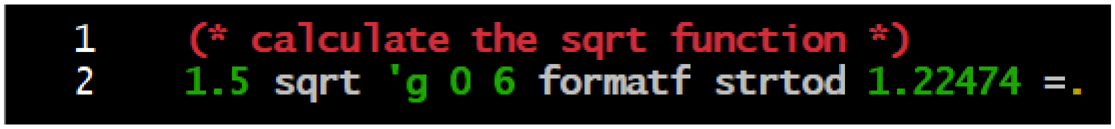
```
(* calculate the sin function *)
0.5 sin 'g 0 6 formatf strtod 0.479426 =.
```

Sin.



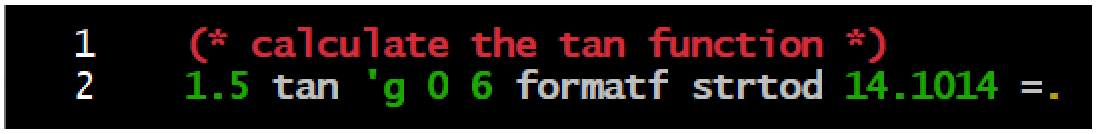
```
(* calculate the sinh function *)
0.5 sinh 'g 0 6 formatf strtod 0.521095 =.
```

Sinh.



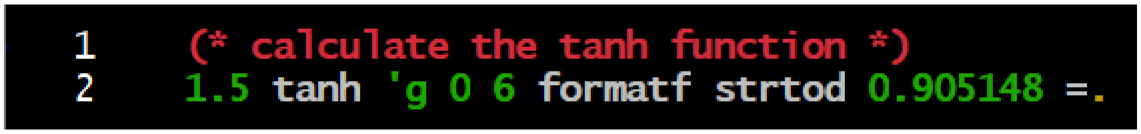
```
(* calculate the sqrt function *)
1.5 sqrt 'g 0 6 formatf strtod 1.22474 =.
```

Sqrt.



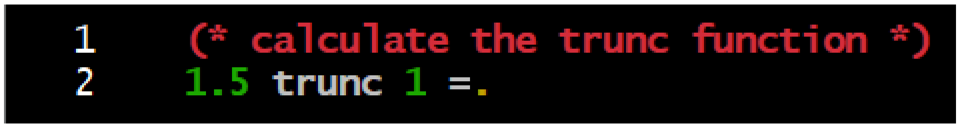
```
(* calculate the tan function *)
1.5 tan 'g 0 6 formatf strtod 14.1014 =.
```

Tan.



```
(* calculate the tanh function *)
1.5 tanh 'g 0 6 formatf strtod 0.905148 =.
```

Tanh.



```
(* calculate the trunc function *)
1.5 trunc 1 =.
```

Trunc.

Builtins

- ⟦log10 ...⟧... F → ⟦...⟧... G
  G is the common logarithm of F.
- ⟦modf ...⟧... F → ⟦...⟧... G H
  G is the fractional part and H is the integer part
  (but expressed as a float) of F.
- ⟦neg ...⟧... I → ⟦...⟧... J
  Integer J is the negative of integer I. Also supports float.
- ⟦sign ...⟧... N1 → ⟦...⟧... N2
  Integer N2 is the sign (-1 or 0 or +1) of integer N1,
  or float N2 is the sign (-1.0 or 0.0 or 1.0) of float N1.
- ⟦sin ...⟧... F → ⟦...⟧... G
  G is the sine of F.
- ⟦sinh ...⟧... F → ⟦...⟧... G
  G is the hyperbolic sine of F.
- ⟦sqrt ...⟧... F → ⟦...⟧... G
  G is the square root of F.
- ⟦tan ...⟧... F → ⟦...⟧... G
  G is the tangent of F.
- ⟦tanh ...⟧... F → ⟦...⟧... G
  G is the hyperbolic tangent of F.
- ⟦trunc ...⟧... F → ⟦...⟧... I
  I is an integer equal to the float F truncated toward zero.

Maintenance

This chapter gives a description of the source code files that are used to build the binary of Moy. The source files are presented alphabetically.

arty.c

Arity calculates the stack effect of a quotation. If the quotation contains a user defined function, the function call is replaced by the body of the function and the body is flagged as used, preventing recursive replacements that would never end. If the quotation contains an anonymous function, that function is searched in the symbol table and the associated arity is processed. Arities contain one or more of the characters: A, D, P, U, N. The end result should be 1. If it isn't then the caller knows that the entire stack needs to be saved before executing the quotation and restored afterwards.

comp.c

This file contains two public functions: `initcompile` and `compileprog`. `Initcompile` is called from the `main` function if the option `-c` is used. Part of the compiling job is delegated to `exitcompile`. `Compileprog` is called from the execute function and replaces the REPL in that funcion. Most of the functionality is delegated to the `compile` function. The core of the compilation is the function `printlist`, that prints a Joy list as the C structures that constitute the list.

eval.c

This file contains `exeterm`, the evaluator of Joy programs. In the case of Moy and Soy, user defined definitions have their body pushed onto the code stack, builtins have their C code executed and all the rest is pushed on the data stack. Primed functions are unprimed when pushed on the data stack. This file also contains functions that access parts of the symbol table.

exec.c

This file contains `execute`, the REPL of the interpreter or `compileprog` in case the program needs to be compiled. This file is also used in compiled programs and that makes compiled programs use a virtual processor instead of a real processor. The advantage of this approach, at least in the case of the Soy runtime is that compiled programs execute the exact same code as interpreted programs. In the case of the Roy runtime, recursion is allowed and that means that part of the builtins are different and need to be tested separately.

globals.h

This file contains all global definitions. Technically there are very few global variables and the ones that are global are not present in this file. This file also includes some other include files. This approach is made possible by CMake that reliably maintains dependencies and recompiles every source file as soon as globals.h or any of the included files is modified.

khash.h

This file contains the hash functions. This file is also used in one of the solutions from the Programming Language Benchmark Game, so it can be assumed that this hashmap implementation has good performance. The interface is sometimes a little odd, having what looks like a function call appearing as L-value.

kvec.h

This file contains generic vectors with the same odd interface that is used in khash.h. When using vectors, the first call must be to `vec_init`, because other vector libraries also require initialization before they can be used. The header is an allocation that is separate from the allocated array. Function `vec_size` can be used with a NULL pointer as parameter.

lexr.l

The function `yylex` replicates the functionality that Joy has in the file scan.c with some additions from 42minjoy that are convenient or necessary in case files need to be compiled. The functionality includes the ability to create a source file listing when reading a file.

main.c

The main file contains `abortexecution` and the start of the program as well as some functions that are called as a consequence of parameter parsing. The main file is also used in compiled programs. Everything that is not needed by compiled programs has been transferred to other files. The main.c in Moy is thus shorter than the ones that are used in Joy and joy1.

modl.c

This file is mostly the same as module.c in Joy. The exception is the interface to `execerror`. In Moy `execerror` has the filename as first parameter, whereas in Joy the filename parameter is passed through a global variable to `execerror`. Among the functions in this file are `savemod` and `undomod` that save and restore global variables in this file. These functions are needed in order to make sure that the double reading of tokens ends up using the same global variables.

otab.c

This file contains the static part of the symbol table. The part that contains the datatypes is in the file itself; all other builtins are included in the data structure with file tabl.c. The file also includes prim.h that contains the declarations of all builtins as well as prim.c that contains definitions of all builtins. The function that gives access to the table is called `readtable`.

parm.c

All parameter checks are in this file and coded with a macro in each of the builtins. This makes it easier to group functions with identical checks under the same entry and the macro makes it easy to disable all checks, as should be done in compiled programs. Those programs are supposed to be thoroughly tested before they are compiled.

pars.y

Joy has a very simple grammar and using a yacc generated parser looks like a bit of an overkill. It so happens that the succinctness of Joy is reflected in the parser and lexical analyzer. The shorter definitions in all three of them can make the meaning of the program a little clearer.

prog.c

This file contains most of the functionality dealing with `env->prog`. Worth mentioning is `prime` that takes care of priming `USR_` and `ANON_FUNCT_` when they are moved from the data stack to the code stack. All other functions are straightforward and avoid duplication in the builtins.

pvec.h

This file contains the functionality required by programs, stacks, and quotations. The normal kvec.h is not used because of the extra bits of information that is needed and polluting the general kvec.h with those bits seems unwanted. Worth mentioning are the void functions that modify existing vectors in a safe way. Their names are made on purpose very long, because what they do looks unsafe.

read.c

This file is a duplication of a part of the parser. It reads factors and terms, triggered by `get` or `fget`. They are needed as long as there is no reentrent yacc parser. As a speciality, they are able to read keywords and thus could be used in a Joy outer interpreter. There is already an inner interpreter in the library.

repl.c

This file contains the symbol table handling that was extracted from the main file as well as the `newnode` function that builds a singleton list that is used in the parser. It was already mentioned that `enteratom` in this file differs from the one in Joy in that it attaches a term to a name after the term has been read. In Joy the entry in the symbol table is created first and only later updated with the body that was read in definition.

save.c

This file implements save and restore of the stack before and after a condition, as required by Joy semantics. If the arity of the condition is assumed to be ok, then no saving/restoring is done. The arity is calculated only once, so repeated execution of the condition cause no extra slowdown. This kind of code is essential in maintaining compatibility with Joy as well as achieving good performance.

scan.c

This file contains some of the functionality that is stored in scan.c in Joy. It can be seen as part of the lexical analyzer, allowing redirection to an included file and also error reporting with file and line where the error occurred as well as the exact position in the line. For the purpose of this error reporting, lines are truncated to INPLINEMAX.

util.c

This file contains a number of utility functions that are used by the lexical analyzer. The contents is completely different from util.c in joy1 and Joy.

writ.c

This file contains the functions \`writefactor\`, \`writeterm\`, and \`writestack\`. Because Moy claims to be stackless, these functions are also written in a way that does not make use of the stack. Noteworthy is that \`writefactor\` writes what looks like Joy source code, but as such the solution is close to 100%. It writes factors in a human-readable format.

xerr.c

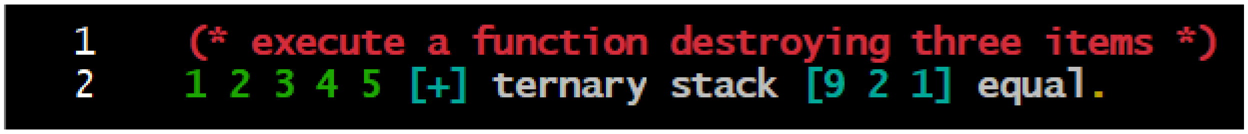
This file contains the function \`execerror\` that was extracted from the main file because it is normally not needed in compiled programs.

ylex.c

This file contains functionality that is needed in order to process some tokens twice. This is the code between HIDE/PRIVATE and END. The functionality is needed so as to allow local definitions as well as public member functions to call each other. The HIDE and MODULE functionality allow some names to be used more than once. Some names, such as \`open\`, \`read\`, \`write\`, \`close\` are very common and if contained in a module, the module makes clear on what object they operate.

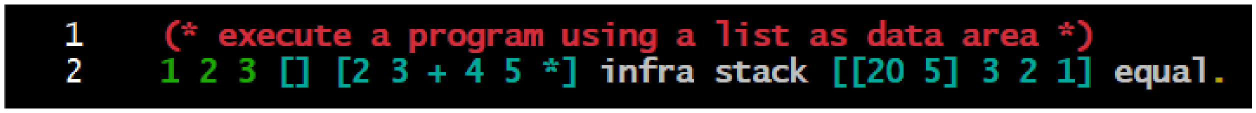
[

Tutorial



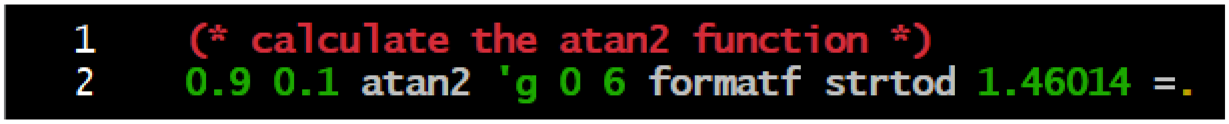
```
(* execute a function destroying three items *)
1 2 3 4 5 [+] ternary stack [9 2 1] equal.
```

Ternary.



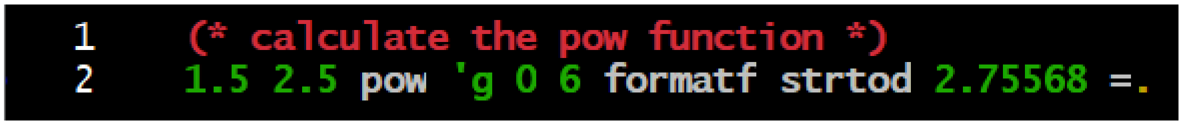
```
(* execute a program using a list as data area *)
1 2 3 [] [2 3 + 4 5 *] infra stack [[20 5] 3 2 1] equal.
```

Infra.



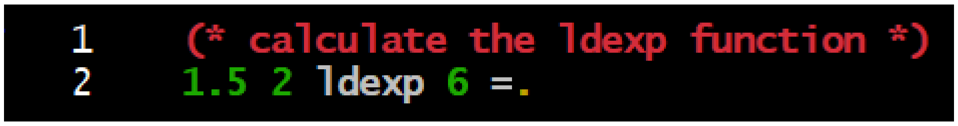
```
(* calculate the atan2 function *)
0.9 0.1 atan2 'g 0 6 formatf strtod 1.46014 =.
```

Atan2.



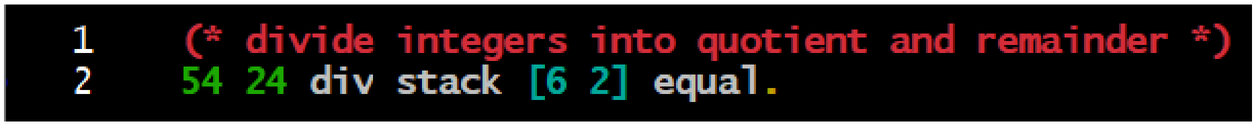
```
(* calculate the pow function *)
1.5 2.5 pow 'g 0 6 formatf strtod 2.75568 =.
```

Pow.



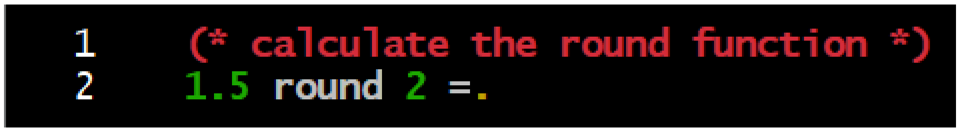
```
(* calculate the ldexp function *)
1.5 2 ldexp 6 =.
```

Ldexp.



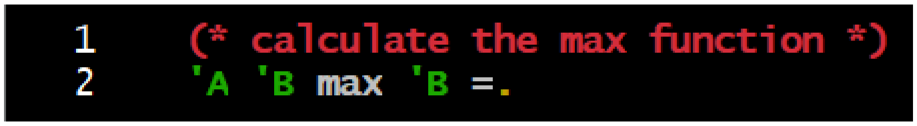
```
(* divide integers into quotient and remainder *)
54 24 div stack [6 2] equal.
```

Div.



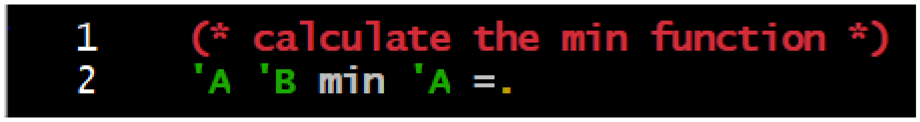
```
(* calculate the round function *)
1.5 round 2 =.
```

Round.



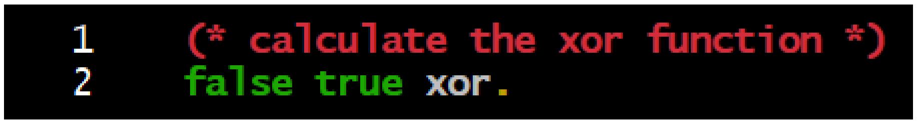
```
(* calculate the max function *)
'A 'B max 'B =.
```

Max.



```
(* calculate the min function *)
'A 'B min 'A =.
```

Min.



```
(* calculate the xor function *)
false true xor.
```

Xor.

Builtins

- ⟦ternary ...⟧... X Y Z [P] → ⟦P ...⟧... R
  Executes P, which leaves R on top of the stack.
  No matter how many parameters this consumes,
  exactly three are removed from the stack.
- ⟦infra ...⟧... L1 [P] → ⟦P ...⟧... L2
  Using list L1 as stack, executes P and returns a new list L2.
  The first element of L1 is used as the top of stack,
  and after execution of P the top of stack becomes the first element of L2.
- ⟦atan2 ...⟧... F G → ⟦...⟧... H
  H is the arc tangent of F / G.
- ⟦pow ...⟧... F G → ⟦...⟧... H
  H is F raised to the Gth power.
- ⟦ldexp ...⟧... F I → ⟦...⟧... G
  G is F times 2 to the Ith power.
- ⟦div ...⟧... I J → ⟦...⟧... K L
  Integers K and L are the quotient and remainder of dividing I by J.
- ⟦round ...⟧... F → ⟦...⟧... G
  [EXT] G is F rounded to the nearest integer.
- ⟦max ...⟧... N1 N2 → ⟦...⟧... N
  N is the maximum of numeric values N1 and N2. Also supports float.
- ⟦min ...⟧... N1 N2 → ⟦...⟧... N
  N is the minimum of numeric values N1 and N2. Also supports float.
- ⟦xor ...⟧... X Y → ⟦...⟧... Z
  Z is the symmetric difference of sets X and Y,
  logical exclusive disjunction for truth values.

Maintenance

Joy and joy1 use a different set of source files. They are listed here, with a small comment on how they differ from Moy.

factor.c

This file contains the code that reads or writes a term or a factor. The code for `readterm` differs between Joy and joy1. The code is also somewhat complicated by the fact that `readfactor` in case of an error does not read a factor.

gc.c

This file is not necessary when linking with the BDW garbage collector. That could be done in the case of the NOBDW version as well. It sounds a little odd to first claim that it is the NOBDW version and then to require that the BDW garbage collector is needed after all. That is why `gc.c` exists. It reduces dependencies on other repositories.

gc.h

This file disables some of the code in `gc.c` that is not needed in the case of Joy. In `globals.h` this file is included with `<gc.h>` meaning that it is first searched in the system include directories. If `gc.c` is used then the compile options should include an `-I.` flag.

globals.h

The purpose of `globals.h` in Joy and joy1 is similar to the one in Moy. There used to be compile options in this file, but they have been either accepted or rejected. There are some new ones in Moy.

interp.c

This file contains the bulk of the interpreter in Joy and joy1. Joy uses macro's to generate C-source code. The header files that are needed for that purpose, as well as the C-source code itself, is included in this file. The old separate compilation techniques of the C compiler are no longer needed.

khash.h

This file is the same as the one in Moy. The legacy version of Joy also used hashing to search the symbol table. This `khash.h` offers a faster solution.

kvec.h

This file is also the same as the one in Moy. The desire was to allow the symbol table to grow when needed. And after this was realized, the memory area as used in Joy could also benefit from a flexible array.

main.c

Compared to Moy, this `main.c` contains more functionality. The symbol table is explained at the start of the file.

module.c

This file is similar to the file `modl.c` in Moy.

scan.c

This file handles reading of source code upto tokens. In case of `HIDE` and `MODULE` tokens need to be listed, allowing them to be read twice. During the first read defined names are read and stored in the symbol table.

utils.c

This file is different between Joy and joy1. It contains the function `newnode` in both of them, but the Joy version also contains the copying garbage collector. This garbage collector is slower than the one in the legacy version, because it uses a non-recursive algorithm during the collection.

ᛉ

Tutorial

```
(* pushes the remainder of the program *)
conts dup equal.
```

Conts.

```
(* identity program, does nothing *)
id.
```

Id.

```
(* converts a symbol to a string *)
[sum] first name "sum" =.
```

Name.

```
(* extract an item from an aggregate at a valid index *)
[4 5 6] 2 at 6 =.
```

At.

```
(* checks whether two lists or items are equal *)
[1 2 3] [1 2 3] equal.
```

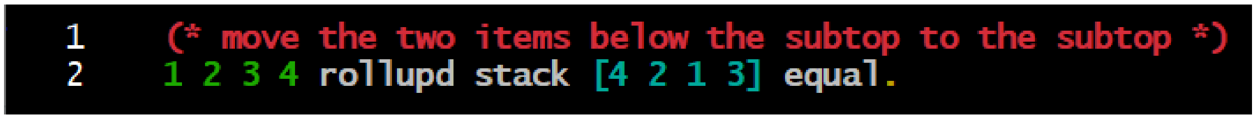
Equal.

```
(* add an element to the front of an aggregate *)
[2 3] 1 swons [1 2 3] equal.
```

Swons.

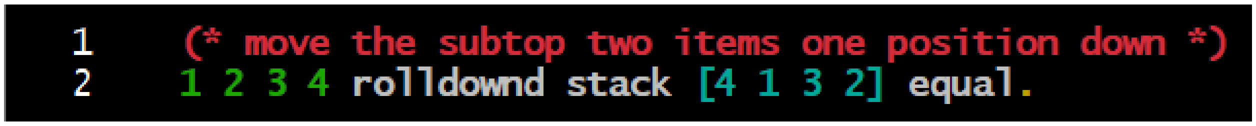
```
(* rotate the top three items *)
1 2 3 rotate stack [1 2 3] equal.
```

Rotate.



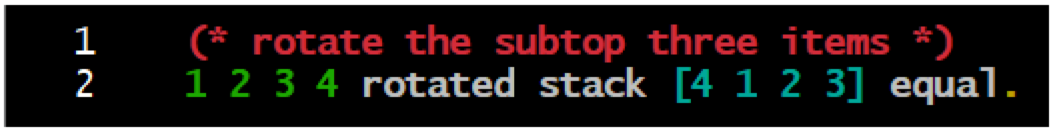
```
(* move the two items below the subtop to the subtop *)
1 2 3 4 rollupd stack [4 2 1 3] equal.
```

Rollupd.



```
(* move the subtop two items one position down *)
1 2 3 4 rolldownd stack [4 1 3 2] equal.
```

Rolldownd.



```
(* rotate the subtop three items *)
1 2 3 4 rotated stack [4 1 2 3] equal.
```

Rotated.

Builtins

- ⟦conts ...⟧... → ⟦...⟧... [[P] [Q] ..]
  Pushes current continuations. Buggy, do not use.
- ⟦id ...⟧... → ⟦...⟧...
  Identity function, does nothing.
  Any program of the form P id Q is equivalent to just P Q.
- ⟦name ...⟧... sym → ⟦...⟧... “sym”
  For operators and combinators, the string "sym" is the name of item sym,
  for literals sym the result string is its type.
- ⟦at ...⟧... A I → ⟦...⟧... X
  X (= A[I]) is the member of A at position I.
- ⟦equal ...⟧... T U → ⟦...⟧... B
  (Recursively) tests whether trees T and U are identical.
- ⟦swons ...⟧... A X → ⟦...⟧... B
  Aggregate B is A with a new member X (first member for sequences).
- ⟦rotate ...⟧... X Y Z → ⟦...⟧... Z Y X
  Interchanges X and Z.
- ⟦rollupd ...⟧... X Y Z W → ⟦...⟧... Z X Y W
  As if defined by: rollupd == [rollup] dip
- ⟦rolldownd ...⟧... X Y Z W → ⟦...⟧... Y Z X W
  As if defined by: rolldownd == [rolldown] dip
- ⟦rotated ...⟧... X Y Z W → ⟦...⟧... Z Y X W
  As if defined by: rotated == [rotate] dip

Maintenance

This chapter gives a rundown of program execution, starting from the main function.

main

The C program starts here. The task of the `main` function is to initialize the garbage collector and to call the real `main` function, named `my_main`. The separation is for the benefit of gc.c that needs to know the top of the call stack. The source code is setup in such a way, that either gc.c or the BDW garbage collector can be used. The latter creates an extra dependency, may not be as portable as gc.c, but is faster.

my_main

The task of `my_main` is to read the command line, initialize global variables, and start the read-eval-print-loop, or REPL for short. Command options are listed in the next chapter.

yyparse

The last action in `my_main` is to call `yyparse`, that initiates the read-eval-print-loop. Within the parser as generated by yacc or bison, only the R and L of REPL are implemented. Eval en print are delegated to `execute`. In Joy, the entire loop is located in `my_main`.

There is not a lot that needs to be said about the grammar, filled with actions. Most actions interact with the symbol table. There is a large action at `USR_`, mostly copied from `readfactor`. One difference: when compiling, the string value of a builtin needs to be remembered, whereas the interpreter wants the function value.

Something else: the sequence `USR_ EQDEF opt_term` is followed by `enteratom`. In this sequence `opt_term` is evaluated before enteratom is executed. In Joy, enteratom is executed as soon as `USR_` is seen and later updated with `opt_term`. That makes a difference in this example, taken from the 42minjoy tutorial:

```
times == dup 0 = [pop pop] [[dup [i] dip] dip pred times] branch.
```

Here, `times` is defined recursively and differs from the builtin in the order of parameters. Combinators normally expect a quotation on the top of the stack, but this old version expects a number on top of the stack. For Joy this is not a problem, because as soon as `times` is seen, it is entered as a new symbol in the symbol table. The `times` in the replacement uses the new version of `times`. For Moy it is a problem: the replacement is processed first and that means that the `times` in the replacement is the old version with a different order of parameters. The solution is to precede the definition above with:

```
        times == ;
```

Now, in both cases, the new version of `times` is used. In C a similar such thing occurred in the joy0 source code:

```
        DMP3→next = newnode(DMP1→op, DMP1→u.num, NULL);
```

Here, it is more or less assumed that `newnode` is executed first and the return value is stored in `DMP3→next` and maybe it was done that way in older compilers. However, the order of evaluation is unspecified. As it happens, `DMP3→next` is evaluated first and when the call to `newnode` triggers the garbage collector, it invalidates `DMP3` and crashes the program. It is a copying garbage collector, that moves all global variables, such as `DMP3` to a different location. The C compiler doesn't know that. The solution is simple: capture the return value of `newnode` in a temporary variable and assign that to `DMP3→next`. Now the C compiler follows the desired order of evaluation. In Joy, calls to newnode are wrapped in a macro `NEWNODE` that takes care of the temporary.

So, Joy has the same unspecified behaviour as the C compiler.

execute

This function calls `exeterm` and then uses the `autoput` setting to print the top of the stack, the whole stack, or only a newline. If there is something on the stack, that is. As part of the read-eval-print-loop, this function takes care of the print and delegates eval to `exeterm`.

exeterm

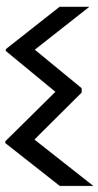
The simplicity of `exeterm` is greatly obscured by the many conditional evaluations that surround it. It switches around the datatypes and takes action accordingly. User defined functions have their body pushed onto the code stack; builtins have their C code executed; all the rest is pushed on the stack. In the case of Moy, two new datatypes have been formulated: `USR_PRIME_` and `ANON_PRIME_`. If an `USR_` or `ANON_FUNCT_` is taken from the stack and pushed onto the code stream, it must be primed in order to prevent its normal behaviour. The primed values are normalized when pushed back on the stack. This is a peculiarity specific to Moy that does not occur in Joy.

builtin

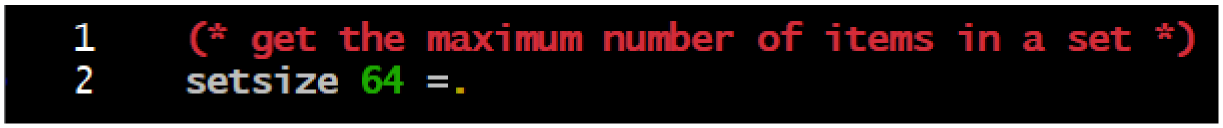
Builtins are called as anonymous functions from `exeterm`. Each one of them has a small explanation in the symbol table. Something more can be said about some of them:

- `app2`, `app3`, `app4` have been marked as Obsolescent. They can be removed, because the documentation now also uses `unary2`, `unary3`, `unary4`. But the joy-in-joy interpreter still uses them;

- some other builtins can als be marked as Obsolescent: \`app1\`, \`app11\`, \`app12\`. Joy uses function composition, not application;
- a number of builtins call other builtins and thus could be entered in a library, for example inilib.joy: \`fold\`, \`enconcat\`, \`fputstring\`, \`condlinrec\`;
- some builtins control the settings of the interpreter and thus are not really part of the language: \`setautoput\`, \`setecho\`, \`setundeferror\`, \`__settracegc\`. They could be command line flags, or some of the command line flags could have been implemented with similar functions;
- \`false\`, \`true\`, \`maxint\` have been marked as IMMEDIATE; the function \`setsize\` could also have been marked as such;
- \`conts\` and \`dump\` are not needed in some implementations, although \`conts\` might actually work in Moy.

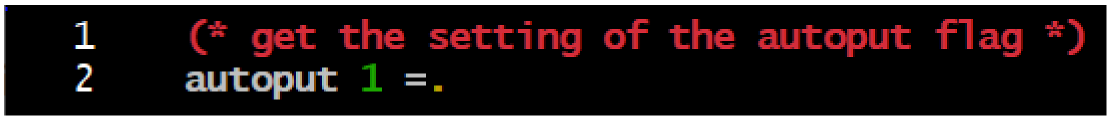


Tutorial



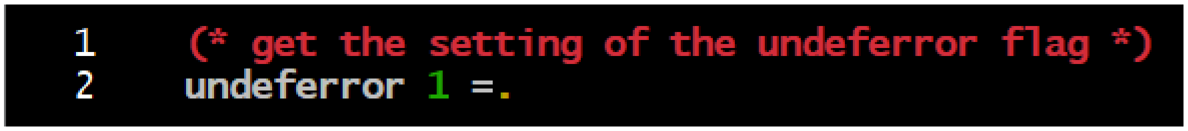
```
(* get the maximum number of items in a set *)
setsize 64 =.
```

Setsize.



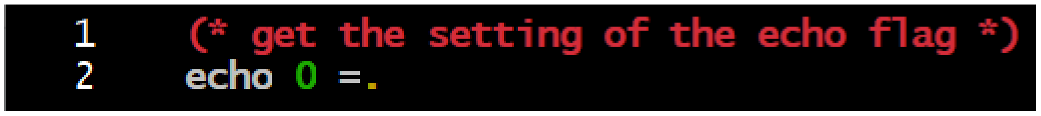
```
(* get the setting of the autoput flag *)
autoput 1 =.
```

Autoput.



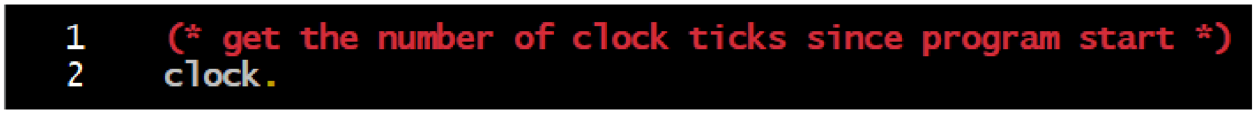
```
(* get the setting of the undeferror flag *)
undeferror 1 =.
```

Undeferror.



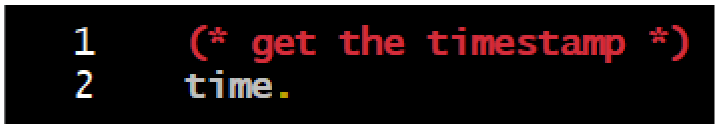
```
(* get the setting of the echo flag *)
echo 0 =.
```

Echo.



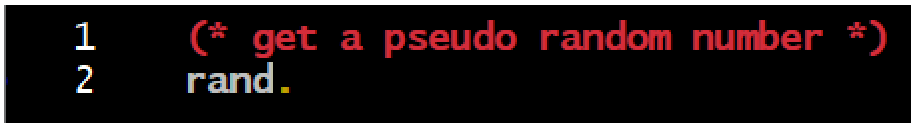
```
(* get the number of clock ticks since program start *)
clock.
```

Clock.



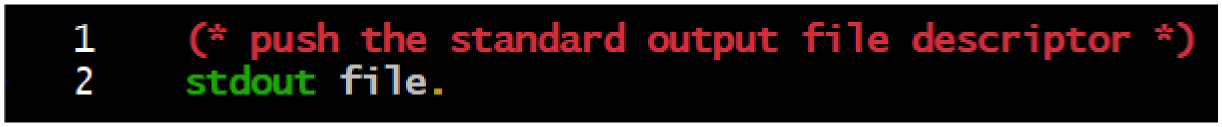
```
(* get the timestamp *)
time.
```

Time.



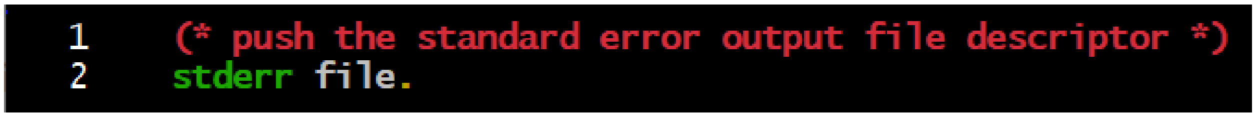
```
(* get a pseudo random number *)
rand.
```

Rand.



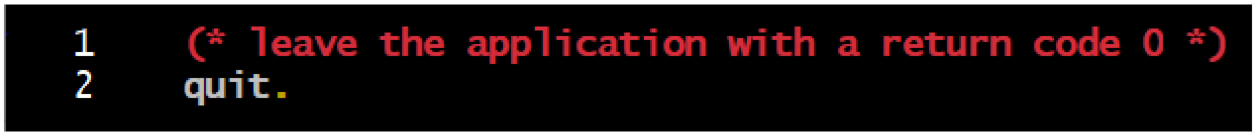
```
(* push the standard output file descriptor *)
stdout file.
```

Stdout.



```
(* push the standard error output file descriptor *)
stderr file.
```

Stderr.



```
(* leave the application with a return code 0 *)
quit.
```

Quit.

Builtins

- ⟦setsize ...⟧... → ⟦...⟧... setsize
  Pushes the maximum number of elements in a set (platform dependent).
  Typically it is 32, and set members are in the range 0..31.
- ⟦autoput ...⟧... → ⟦...⟧... I
  Pushes current value of flag for automatic output, I = 0..2.
- ⟦undeferror ...⟧... → ⟦...⟧... I
  Pushes current value of undefined-is-error flag.
- ⟦echo ...⟧... → ⟦...⟧... I
  Pushes value of echo flag, I = 0..3.
- ⟦clock ...⟧... → ⟦...⟧... I
  [IMPURE] Pushes the integer value of current CPU usage in milliseconds.
- ⟦time ...⟧... → ⟦...⟧... I
  [IMPURE] Pushes the current time (in seconds since the Epoch).
- ⟦rand ...⟧... → ⟦...⟧... I
  [IMPURE] I is a random integer.
- ⟦stdout ...⟧... → ⟦...⟧... S
  [FOREIGN] Pushes the standard output stream.
- ⟦stderr ...⟧... → ⟦...⟧... I
  [FOREIGN] Pushes the standard error stream.
- ⟦quit ...⟧... → ⟦...⟧...
  [IMPURE] Exit from Joy.

Maintenance

Joy and joy1 use a code base that is different from that of Moy and deserve separate treatment. They are maintained in order to keep the language the same among the three implementations.

main

The `main` function in Joy and joy1 is the same as in Moy.

my_main

The `my_main` function in Joy and joy1 is similar to the one in Moy. The REPL is inlined, there are fewer command-line options, and option `-h` is less elaborate.

exeterm

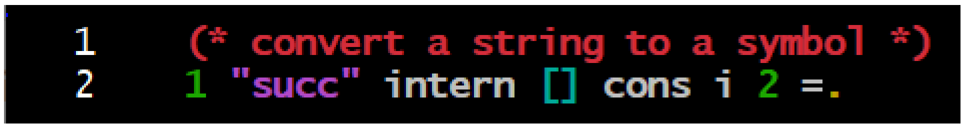
In Joy and joy1 `exeterm` is called from the `my_main` function. This function can be considered the virtual processor of the Joy virtual machine and unlike real processors it can be called recursively.

builtin

Builtins in Joy and joy1 have the same functionality as in Moy, even though the coding is different. The code of some builtins in Joy differs from the ones in joy1 because of the use of dump1-5.

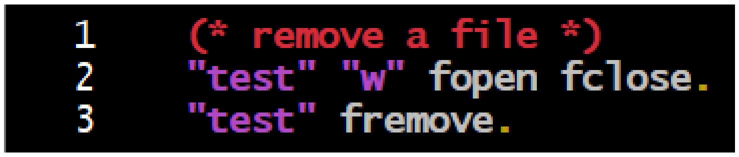
Builtins that have been added since the legacy version are marked with `[EXT]` in the symbol table. No such marking is present to distinguish the legacy version from joy0 and no comparison is made with 42minjoy.

Tutorial



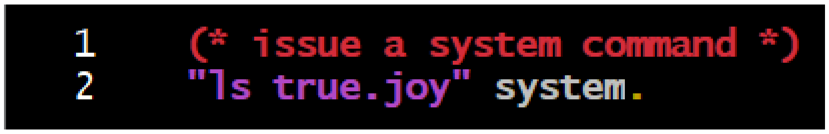
```
(* convert a string to a symbol *)
1 "succ" intern [] cons i 2 =.
```

Intern.



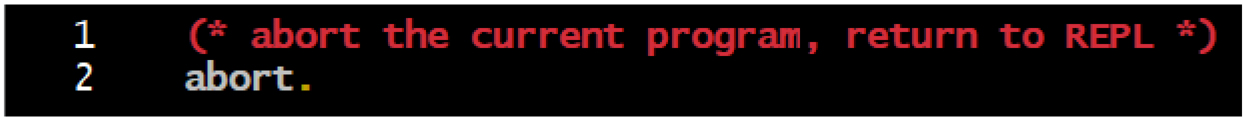
```
(* remove a file *)
"test" "w" fopen fclose.
"test" fremove.
```

Fremove.



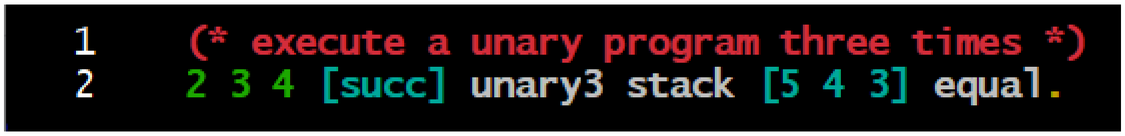
```
(* issue a system command *)
"ls true.joy" system.
```

System.



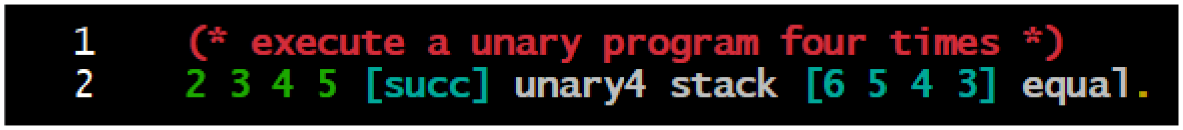
```
(* abort the current program, return to REPL *)
abort.
```

Abort.



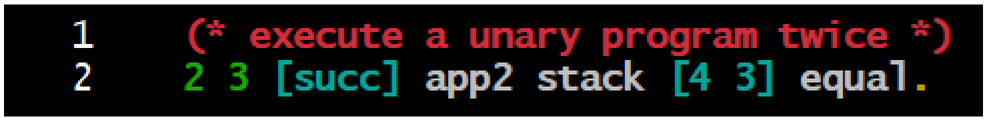
```
(* execute a unary program three times *)
2 3 4 [succ] unary3 stack [5 4 3] equal.
```

Unary3.



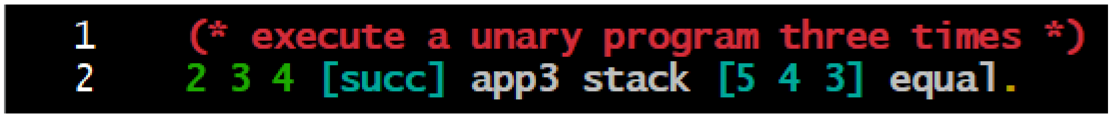
```
(* execute a unary program four times *)
2 3 4 5 [succ] unary4 stack [6 5 4 3] equal.
```

Unary4.



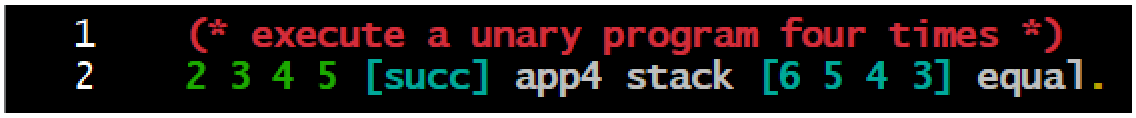
```
(* execute a unary program twice *)
2 3 [succ] app2 stack [4 3] equal.
```

App2.



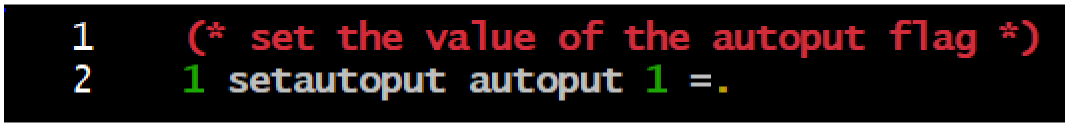
```
(* execute a unary program three times *)
2 3 4 [succ] app3 stack [5 4 3] equal.
```

App3.



```
(* execute a unary program four times *)
2 3 4 5 [succ] app4 stack [6 5 4 3] equal.
```

App4.



```
(* set the value of the autoput flag *)
1 setautoput autoput 1 =.
```

Setautoput.

Builtins

- ⟦intern ...⟧... “sym” → ⟦...⟧... sym
  Pushes the item whose name is "sym".
- ⟦fremove ...⟧... P → ⟦...⟧... B
  [FOREIGN] The file system object with pathname P is removed from the file system. B is a boolean indicating success or failure.
- ⟦system ...⟧... “command” → ⟦...⟧...
  [IMPURE] Escapes to shell, executes string "command".
  The string may cause execution of another program.
  When that has finished, the process returns to Joy.
- ⟦abort ...⟧... → ⟦...⟧...
  Aborts execution of current Joy program, returns to Joy main cycle.
- ⟦unary3 ...⟧... X1 X2 X3 [P] → ⟦...⟧... R1 R2 R3
  Executes P three times, with Xi, returns Ri (i = 1..3).
- ⟦unary4 ...⟧... X1 X2 X3 X4 [P] → ⟦...⟧... R1 R2 R3 R4
  Executes P four times, with Xi, returns Ri (i = 1..4).
- ⟦app2 ...⟧... X1 X2 [P] → ⟦...⟧... R1 R2
  Obsolescent. == unary2
- ⟦app3 ...⟧... X1 X2 X3 [P] → ⟦...⟧... R1 R2 R3
  Obsolescent. == unary3
- ⟦app4 ...⟧... X1 X2 X3 X4 [P] → ⟦...⟧... R1 R2 R3 R4
  Obsolescent. == unary4
- ⟦setautoput ...⟧... I → ⟦...⟧...
  [IMPURE] Sets value of flag for automatic put to I (if I = 0, none;
  if I = 1, put; if I = 2, stack).

Maintenance

This chapter gives an overview of the many ways that the source files can be compiled, the so called conditional compilations.

ALARM

It is possible to set a time limit, such as -DALARM=60, in order to satisfy restrictions imposed on solutions submitted to projectEuler. That website presents mathematical challenges that can be solved by programming a computer and restricts solutions to one minute.

YYDEBUG

When activated, this define allows the user to see the parse tree, as maintained by bison. A \`-y\` command flag is also required.

VERS

This define can be used to tell something about the compile options, such as whether it is a Release build, and what the version number is. The version happens to be always 1.0 and whether it is a Release build or not can also be seen by executing \`ccmake .\`. That program reads the latest CMakeCache.txt and reports about the options available therein.

NCHECK

This, when defined, turns off all runtime checks. It is not recommended to compile the source without runtime checks. There are some checks that are open to discussion, such as that in \`plus\` and \`minus\`. It is allowed to add an integer to a character. It is not allowed to add a character to an integer, even though it could be useful, for example in the following program:

```
5 "test" [+] map.
```

The 5 needs to be pushed only once, but this construct is rejected by the type checker. As it happens, the \`[+]\` is also rejected by the arity check.

USE_MULTI_THREADS_JOY

Multitasking needs to be enabled in globals.h and it remains to be seen whether it is a good addition to the capabilities of Moy. It does make the M in Moy acquire its intended meaning.

USE_BIGNUM_ARITHMETIC

Bignums also need to be enabled in globals.h and they are definitely a good addition, although at this moment not finished.

BYTECODE

The compiler can be enabled with this flag and of course the files that implement the compiler must be present.

NDEBUG

This define is enabled by CMake when compiling in Release mode. It disables all assert statements. As it is, there are currently no assert statements in the source code.

DEBUG

All debugging with printf was enabled when this define was present. The printf statements under debugging have been removed as they clutter the code. They are still present in joy0 in case anyone cares.

NOBDW

The NOBDW version of Joy needs this define, because Joy shares source code with joy1, except that some source code is different, thanks to this define.

TRACEGC

The NOBDW version of Joy also has the ability to debug the garbage collector. That facility has not been removed. The debugging can be turned on and off from within the language.

_MSC_VER

Special instructions on behalf of the Microsoft C compiler are guarded by these defines. Present in `khash.h` and `pars.c`.

SOURCE_DATE_EPOCH

This shell variable, when available during the compilation of main.c causes the date and time stamp to be that of the last known version created by the author of Joy. This timestamp indicates that the language is still the one from that date, even though some builtins have been added since.

#if 0

Some code is disabled with this define. It is code that is not executed and still needs to be kept around in case of future changes.

B

Tutorial

```
(* select a line based on the value of the second parameter *)
1 [[1 "one"]
   [2 "two"]
   ["other"]] case "one" =.
```

Case.

```
(* execute a line based on the value of the second parameter *)
1 [[[1 =] "one"]
   [[2 =] "two"]
   ["other"]] cond "one" =.
```

Cond.

```
(* tell whether some elements of an aggregate satisfy a condition *)
[1 2 3] [2 <] some.
```

Some.

```
(* tell whether all elements of an aggregate satisfy a condition *)
[1 2 3] [4 <] all.
```

All.

```
(* retrieve the value of an environment variable *)
"PATH" getenv.
```

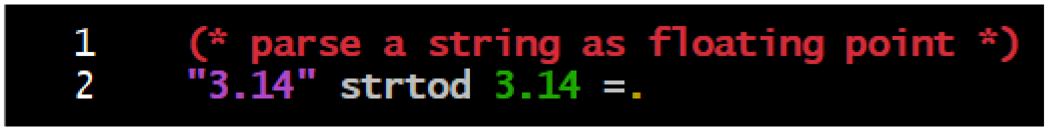
Getenv.

```
(* see whether an element is member of an aggregate *)
[1 2 3] 2 has.
```

Has.

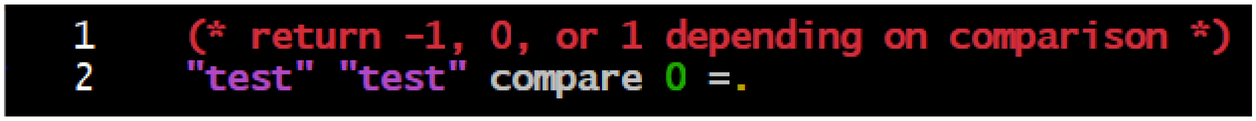
```
(* see whether an element is member of an aggregate *)
2 [1 2 3] in.
```

In.



```
(* parse a string as floating point *)
"3.14" strtod 3.14 =.
```

Strtod.



```
(* return -1, 0, or 1 depending on comparison *)
"test" "test" compare 0 =.
```

Compare.

Builtins

- ⟦case ...⟧... X [...[X Y]...] → ⟦Y...⟧...
  Indexing on the value of X, execute the matching Y.
- ⟦cond ...⟧... [...[[Xi] Ti]...[D]] → ⟦Ti ...⟧...
  Tries each Bi. If that yields true, then executes Ti and exits.
  If no Bi yields true, executes default D.
- ⟦some ...⟧... A [B] → ⟦A_0 B if-true-jump-to-(1) ... A_n B (1) ...⟧… X
  Applies test B to members of aggregate A, X = true if some pass.
- ⟦all ...⟧... A [B] → ⟦A_0 B if-false-jump-to-(1) ... A_n B (1) ...⟧… X
  Applies test B to members of aggregate A, X = true if all pass.
- ⟦getenv ...⟧... “variable” → ⟦...⟧… "value”
  Retrieves the value of the environment variable "variable".
- ⟦has ...⟧... A X → ⟦...⟧… B
  Tests whether aggregate A has X as a member.
- ⟦in ...⟧... X A → ⟦...⟧… B
  Tests whether X is a member of aggregate A.
- ⟦strtod ...⟧... S → ⟦...⟧… R
  String S is converted to the float R.
- ⟦compare ...⟧... A B → ⟦...⟧… I
  I (=-1,0,+1) is the comparison of aggregates A and B.
  The values correspond to the predicates <=, =, >=.

Maintenance

This chapter plays the speed game, a very difficult game.

Implementation	Timing fib(40)
42minjoy	1m6
Joy0	34
Legacy	34
Joy	1m53
Joy1	3m7
Moy	2m15
Soy	1m41
Roy	27
Foy	2m8

Every benchmark wants to prove a point and this benchmark is no exception. The point to prove is that Moy is not the slowest implementation. As can be seen from the table joy1 is slowest.

The source code that is used by 42minjoy comes in two files, a library file and a program file. The library file contains:

```
fib == dup 2 < [[1 - dup fib swap 1 - fib +] []] of i.
```

The program file contains:

```
40 fib.
```

The Fibonacci program of the other implementations is:

```
40 [small] [] [pred dup pred] [+] binrec.
```

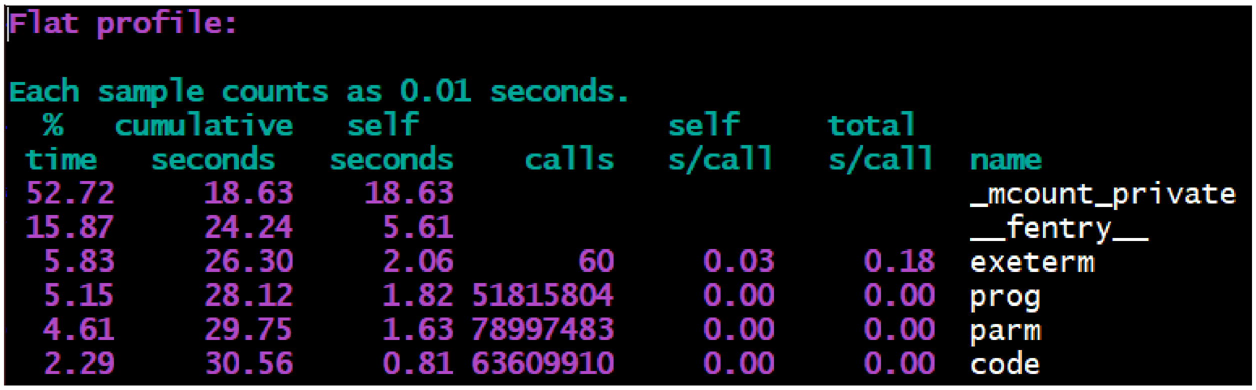
Except that Moy, Soy, Roy, Foy all have `dup small` instead of small.

This game can be played by replacing “CC = gcc” in the makefile with “CC = gcc -pg.”

Then:

```
make clean
make
cd lib
../joy grmtst.joy
gprof ../joy >t
vim t
```

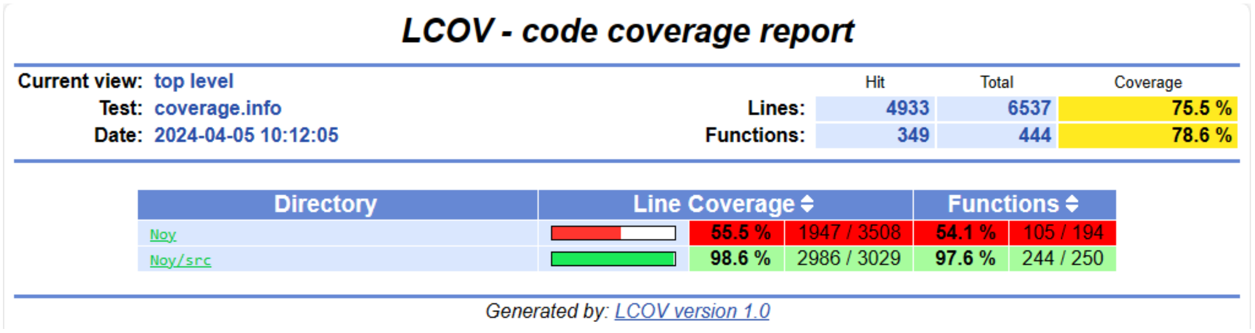
That gives the picture on the next page. The picture can be inspected to see if there is anything that can be improved.



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 52.72     18.63    18.63                             _mcount_private
 15.87     24.24     5.61                             __fentry__
  5.83     26.30     2.06       60     0.03     0.18  exeterm
  5.15     28.12     1.82 51815804     0.00     0.00  prog
  4.61     29.75     1.63 78997483     0.00     0.00  parm
  2.29     30.56     0.81 63609910     0.00     0.00  code
```

The call to the parm-function can be prevented by compiling with -DNCHECK. That would save 1.63 seconds.

There is another kind of game that can be played. It is called profiling and is used to verify that all functions are indeed executed at least once. This does not guarantee that programs are without defects, but it does give some assurance.



LCOV - code coverage report

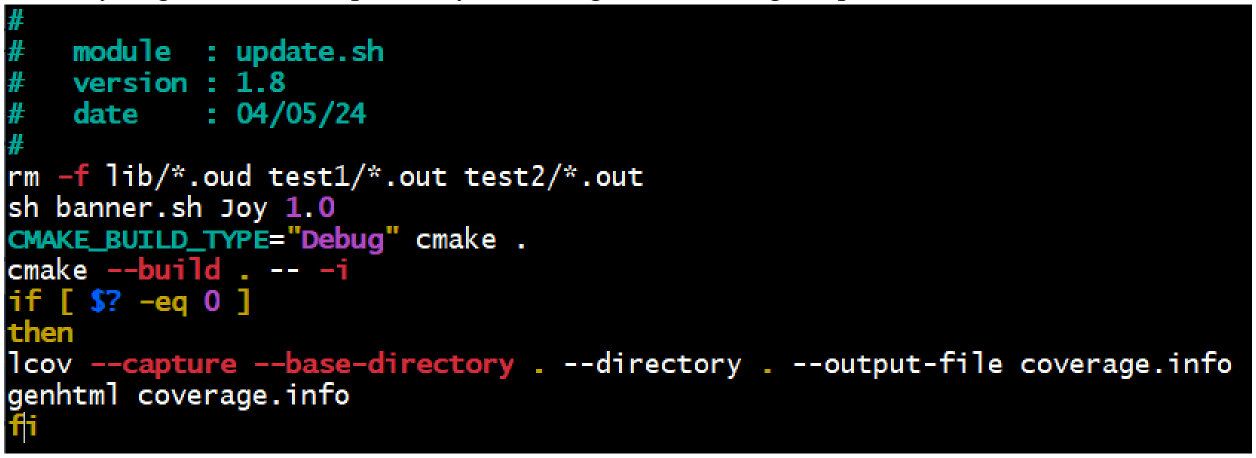
				Hit	Total	Coverage
Current view:	top level					
Test:	coverage.info		Lines:	4933	6537	75.5 %
Date:	2024-04-05 10:12:05		Functions:	349	444	78.6 %

Directory	Line Coverage ⬍			Functions ⬍	
Noy		55.5 %	1947 / 3508	54.1 %	105 / 194
Noy/src		98.6 %	2986 / 3029	97.6 %	244 / 250

Generated by: LCOV version 1.0

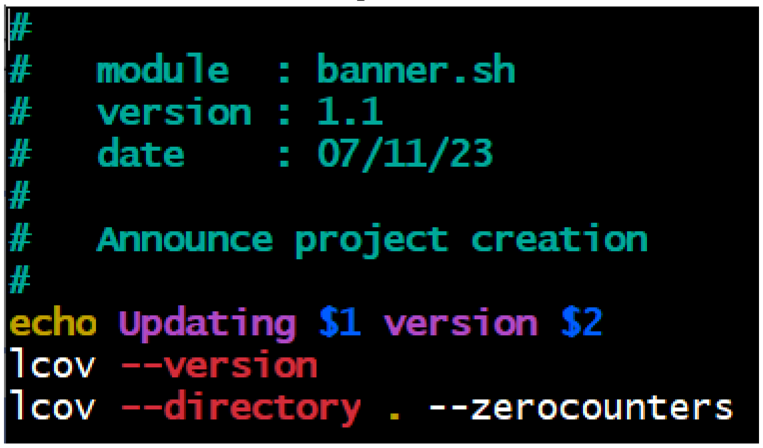
The src-directory should display 100%. It currently doesn't because multi-tasking is not tested and some other lines are also not executed.

The way to generate this report is by executing the following script:



```
#
#   module  : update.sh
#   version : 1.8
#   date    : 04/05/24
#
rm -f lib/*.oud test1/*.out test2/*.out
sh banner.sh Joy 1.0
CMAKE_BUILD_TYPE="Debug" cmake .
cmake --build . -- -i
if [ $? -eq 0 ]
then
lcov --capture --base-directory . --directory . --output-file coverage.info
genhtml coverage.info
fi
```

The script makes use of banner.sh, another script:

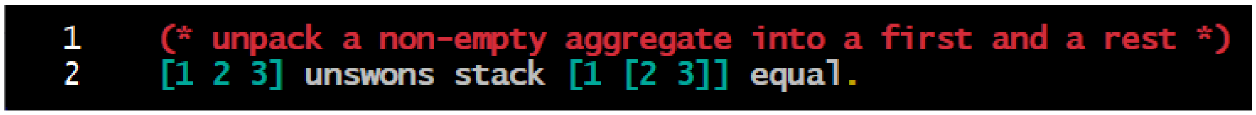


```
#
#   module  : banner.sh
#   version : 1.1
#   date    : 07/11/23
#
#   Announce project creation
#
echo Updating $1 version $2
lcov --version
lcov --directory . --zerocounters
```

Profiling is useful for testing. It can also be used to discover whether there are any lines of code that are executed more often than expected.

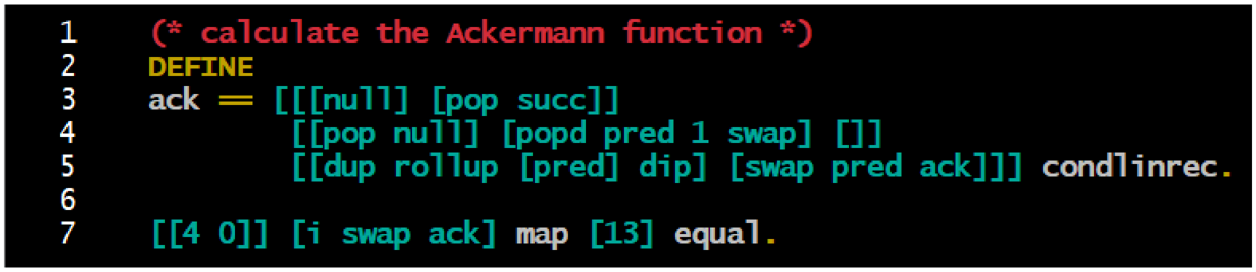
M

Tutorial



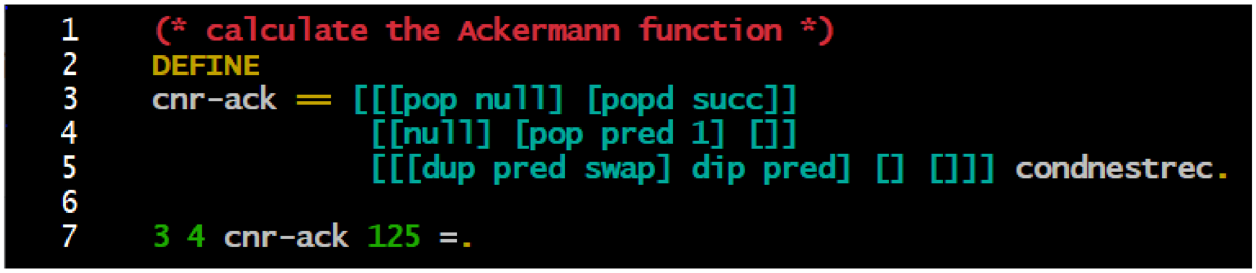
```
(* unpack a non-empty aggregate into a first and a rest *)
[1 2 3] unswons stack [1 [2 3]] equal.
```

Unswons.



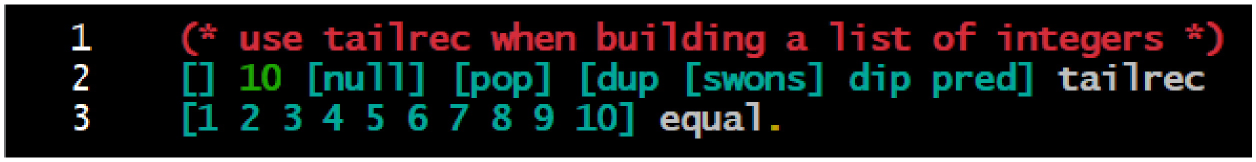
```
(* calculate the Ackermann function *)
DEFINE
ack == [[[null] [pop succ]]
        [[pop null] [popd pred 1 swap] []]
        [[dup rollup [pred] dip] [swap pred ack]]] condlinrec.

[[4 0]] [i swap ack] map [13] equal.
```

Condlinrec.



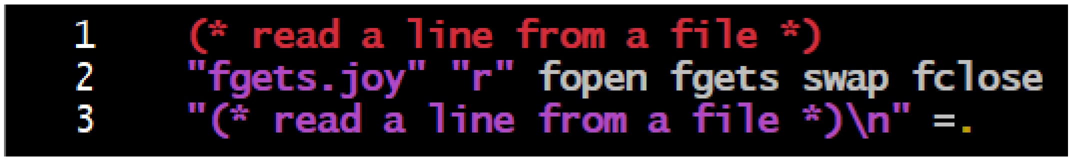
```
(* calculate the Ackermann function *)
DEFINE
cnr-ack == [[[pop null] [popd succ]]
            [[null] [pop pred 1] []]
            [[[dup pred swap] dip pred] [] []]] condnestrec.

3 4 cnr-ack 125 =.
```

Condnestrec.



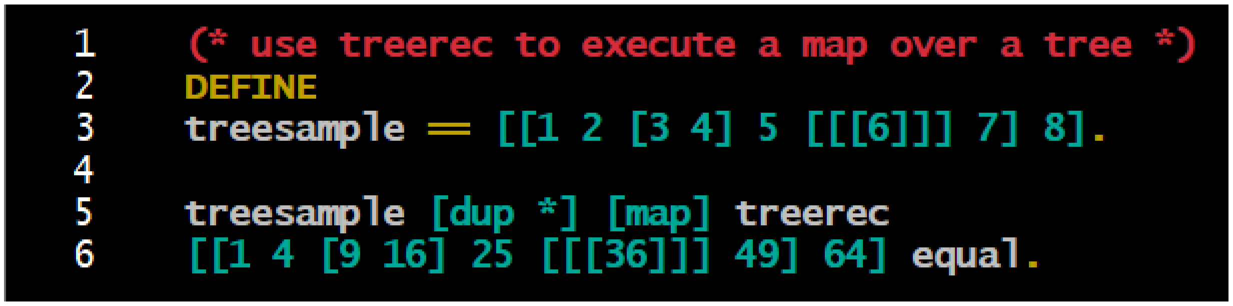
```
(* use tailrec when building a list of integers *)
[] 10 [null] [pop] [dup [swons] dip pred] tailrec
[1 2 3 4 5 6 7 8 9 10] equal.
```

Tailrec.



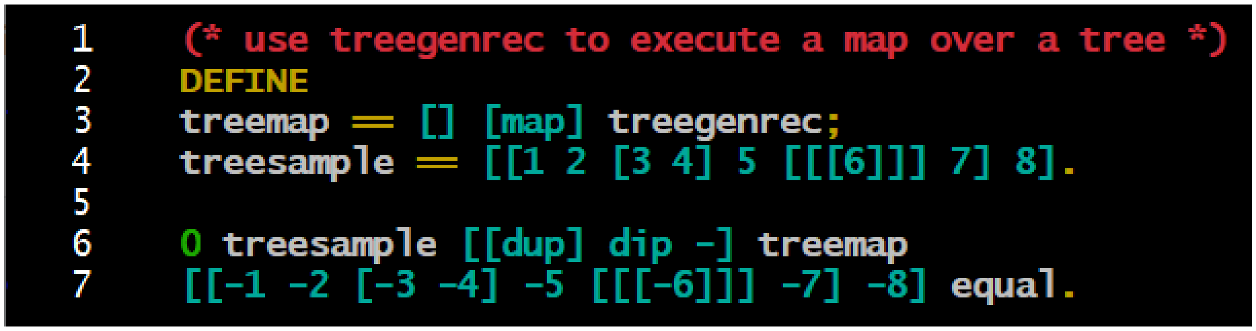
```
(* read a line from a file *)
"fgets.joy" "r" fopen fgets swap fclose
"(* read a line from a file *)\n" =.
```

Fgets.



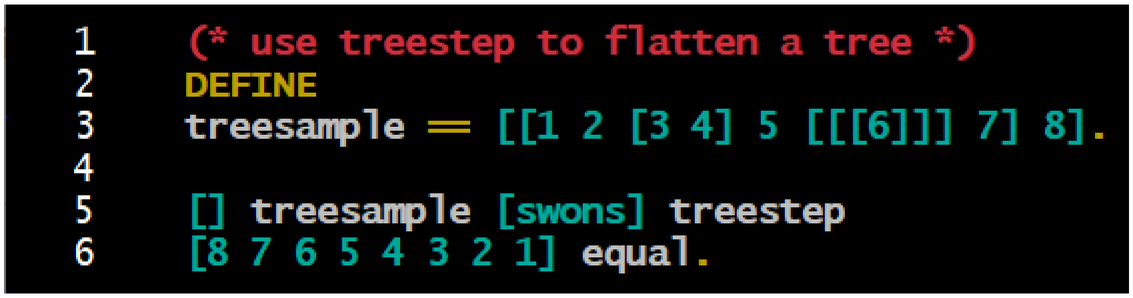
```
(* use treerec to execute a map over a tree *)
DEFINE
treesample == [[1 2 [3 4] 5 [[[6]]] 7] 8].

treesample [dup *] [map] treerec
[[1 4 [9 16] 25 [[[36]]] 49] 64] equal.
```

Treerec.



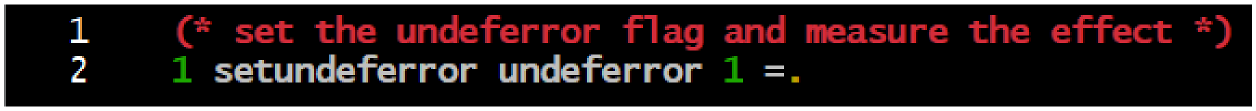
```
(* use treegenrec to execute a map over a tree *)
DEFINE
treemap == [] [map] treegenrec;
treesample == [[1 2 [3 4] 5 [[[6]]] 7] 8].

0 treesample [[dup] dip -] treemap
[[-1 -2 [-3 -4] -5 [[[-6]]] -7] -8] equal.
```

Treegenrec.



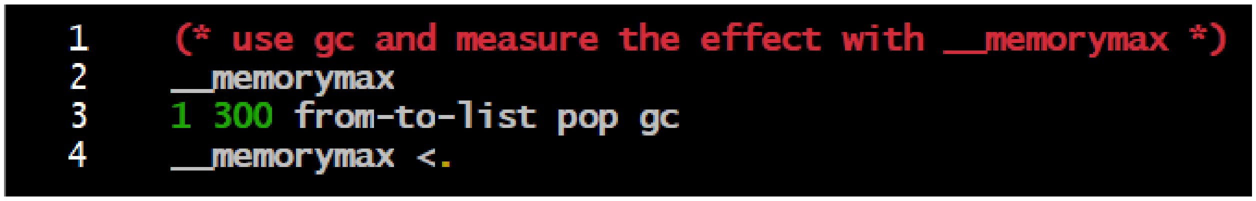
```
(* use treestep to flatten a tree *)
DEFINE
treesample == [[1 2 [3 4] 5 [[[6]]] 7] 8].

[] treesample [swons] treestep
[8 7 6 5 4 3 2 1] equal.
```

Treestep.



```
(* set the undeferror flag and measure the effect *)
1 setundeferror undeferror 1 =.
```

Setundeferror.



```
(* use gc and measure the effect with __memorymax *)
__memorymax
1 300 from-to-list pop gc
__memorymax <.
```

Gc.

Builtins

- ⟦unswons ...⟧... A → ⟦...⟧... R F
  R and F are the rest and the first of non-empty aggregate A.
- ⟦condlinrec ...⟧... [[C1] [C2] ... [D]] → ⟦...⟧...
  Each [Ci] is of the form [[B] [T]] or [[B] [R1] [R2]].
  Tries each B. If that yields true and there is just a [T], executes T and exit.
  If there are [R1] and [R2], executes R1, recurses, executes R2.
  Subsequent case are ignored. If no B yields true, then [D] is used.
  It is then of the form [[T]] or [[R1] [R2]]. For the former, executes T.
  For the latter executes R1, recurses, executes R2.
- ⟦condnestrec ...⟧... [[C1] [C2] ... [D]] → ⟦...⟧...
  A generalisation of condlinrec.
  Each [Ci] is of the form [[B] [R1] [R2] .. [Rn]] and [D] is of the form
  [[R1] [R2] .. [Rn]]. Tries each B, or if all fail, takes the default [D].
  For the case taken, executes each [Ri] but recurses between any two
  consecutive [Ri] (n > 3 would be exceptional.)
- ⟦tailrec ...⟧... [P] [T] [R1] → ⟦...⟧...
  Executes P. If that yields true, executes T.
  Else executes R1, recurses.
- ⟦fgets ...⟧... S → ⟦...⟧... S L
  [FOREIGN] L is the next available line (as a string) from stream S.
- ⟦treerec ...⟧... T [O] [C] → ⟦...⟧...
  T is a tree. If T is a leaf, executes O. Else executes [[[O] C] treerec] C.
- ⟦treegenrec ...⟧... T [O1] [O2] [C] → ⟦...⟧...
  T is a tree. If T is a leaf, executes O1.
  Else executes O2 and then [[[O1] [O2] C] treegenrec] C.
- ⟦treestep ...⟧... T [P] → ⟦...⟧...
  Recursively traverses leaves of tree T, executes P for each leaf.
- ⟦setundeferror ...⟧... I → ⟦...⟧...
  [IMPURE] Sets flag that controls behavior of undefined functions
  (0 = no error, 1 = error).

- ⟦gc ...⟧... → ⟦...⟧...
  [IMPURE] Initiates garbage collection.

Maintenance

The following modifications were done to the original sources:

- Integers and sets are 64-bit;
- On reading, integers that are too large are converted to double;
- Symbol table, memory area, tokenlist, symbols and include directories can grow when needed;
- Local symbols and public member functions can call each other;
- ~~Modules within other modules are in the global namespace~~;
- Library files can be stored anywhere;
- Line numbering is resumed after including a file;
- Comparison of values is centralized in one Compare function;
- Additional builtins are marked with [EXT], [MTH], [NUM];
- Parameter checks are centralized;
- Strings are garbage collected.

integers

Integers are now 64-bit. This differs from the legacy version, that uses 32-bit integers.

conversion

The documentation, called the current implementation, has been updated to reflect the change that integers that are too large on input are silently converted to floating point with loss of precision. In case bignums have been activated in Moy, conversion is to bignum with loss of computation speed. Changes to the documentation, other than correcting typos, are done by inserting annotations.

symbols

Symbol table and memory can grow when needed. The legacy version had the symbol table set at a maximum of 1000. If that is not enough, it needs to be increased and the Joy binary needs to be rebuilt. Likewise, in Joy, the maximum size of memory was set to 20000. If that is not enough, and that happened with the mandelbrot program, it needs to be increased. It is nicer when the program automatically adjusts these maximum sizes without recompilation.

locals

The only way this could be realized is by reading the code between HIDE and IN and MODULE and END twice; during the first read the symbols that are declared are entered in the symbol table and during the second read the declarations are added to the symbols. The only way source code can be read twice is by using a buffer that collects the tokens during the first read and reading tokens from the buffer during the second read. Reading source code directly is not possible because the code may come from stdin and seeking on stdin is not guaranteed to be possible. Also, there could be a switch of files between HIDE and IN or between MODULE and END. In that case, seeking is also not possible.

modules

Modules are present in the global namespace because of the way they are stored in the symbol table. Modules are comparable to structures in C. Structures are also in the global namespace, even when defined within other structures. As Joy is built in C some of the syntax and semantics of C are also present in Joy. That is why this behaviour is preferable. The behaviour of the legacy version can be restored. All it takes is a hidden flag in every symbol table entry and the following changes to the code:

- At the start of a module, the size of the symbol table is registered. At the end of the module, all symbol table entries between the old size and the new size are inspected to see whether they refer to a module that is different from the module that is about to finish. If so, these module entries are marked as hidden.
- When reading a module.member, the hidden field is inspected. If true, the module.member is reported as "not found".

The legacy behaviour can be restored, but there is no compelling reason to do so. As long as the implementation language is C, it should be as it is now. Ok, the behaviour of the legacy version is restored. Not in Moy and also only at top level.

library

Libraries are important. Not all functionality that is needed needs to be implemented in C. Some of it can be implemented in Joy just as well. That is what the libraries are used for. Placing \`usrlib.joy\` in the current directory is good enough and from there other libraries can be loaded, such as \`inilib.joy\`. This \`inilib.joy\` contains a function \`libload\` that can load other libraries. In a distribution where the Joy binary is located in a bin-directory and the libraries in a \`lib-directory\` it would be very convenient if \`libload\` in \`inilib.joy\` would support such a setup. It does now.

lines

The documentation promised: "When input reverts to an earlier file, the earlier line numbering is resumed." but that is not what happens in the legacy version. It did happen in 42minjoy and was restored in Joy and Moy.

comparisons

Comparisons of all datatypes are centralized in the file `compare.h`. The file `grmtst.joy` shows some earlier attempts to reconcile the way that symbols are compared in `in` and `has`. The legacy version compares only the `num` field, disregarding the datatype, considering a match if the `num` fields are bitwise the same. In all other comparisons symbols and strings are treated equal if they look the same. These divergent ideas about equality is not acceptable. That is why the `Compare` function was created. This created a problem when processing `grmtst.joy`. Considering the symbol `*` the same as the string `"*"` is what caused the problem. Same for `+` and `"+"`. The solution is not to look at what the symbol looks like, but how it is pronounced: `*` is pronounced `mul` or `ast` and then differs from the looks of the string `"*"`.

builtins

The language can be extended with new datatypes and new builtins without change to the language itself. New datatypes occur in Moy and are inserted after `FILE_`. `FLOAT_` and `FILE_` in the legacy version are new additions compared to joy0. New builtins are marked with `[EXT]`, `[MTH]`, or `[NUM]`. The last two groups of builtins are only available in Moy and only when they have been enabled in `globals.h`. The repository must then also include the files that implement these builtins.

parameters

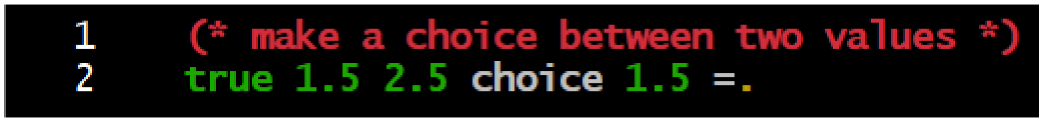
Moy has parameter checks centralized in file `parm.c`. This is convenient when compiling Joy. The file `parm.c` need not be linked into compiled programs. This file is not present in Joy and joy1, because these implementations already have parameter checks centralized in `interp.c` and replacing those with `parm.c` risks the danger of incompatibility with the legacy version.

strings

Strings were already garbage collected in joy1, but Joy also needed an extra garbage collector for this to happen. The BDW garbage collector is preferred, because it is faster, but if not many strings are allocated, `gc.c` will do just fine. Besides, first claiming that Joy is the NOBDW version and then insisting that the BDW should be used after all, looks a bit odd. The BDW is faster than gc.c.

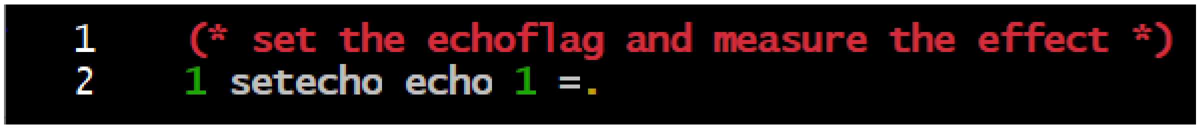
ᛗ

Tutorial



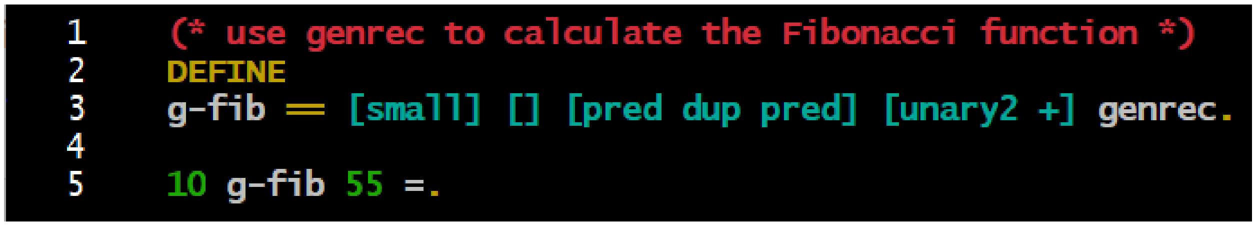
```
(* make a choice between two values *)
true 1.5 2.5 choice 1.5 =.
```

Choice.



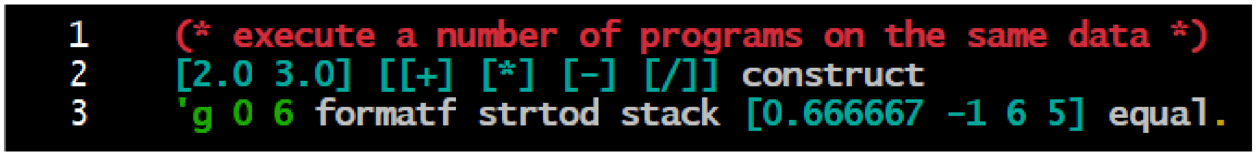
```
(* set the echoflag and measure the effect *)
1 setecho echo 1 =.
```

Setecho.



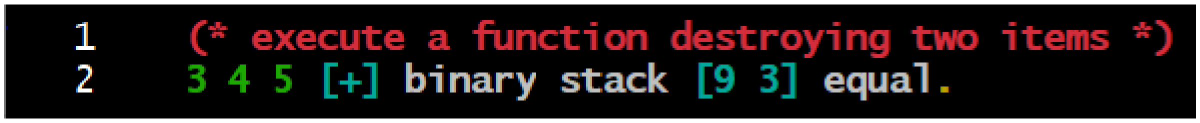
```
(* use genrec to calculate the Fibonacci function *)
DEFINE
g-fib == [small] [] [pred dup pred] [unary2 +] genrec.

10 g-fib 55 =.
```

Genrec.



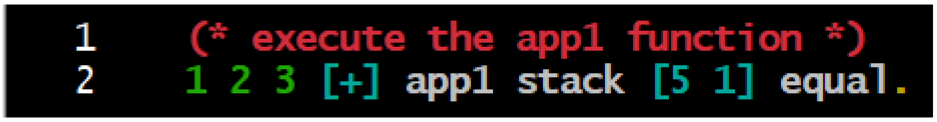
```
(* execute a number of programs on the same data *)
[2.0 3.0] [[+] [*] [-] [/]] construct
'g 0 6 formatf strtod stack [0.666667 -1 6 5] equal.
```

Construct.



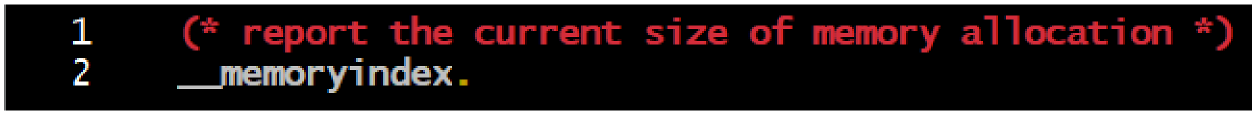
```
(* execute a function destroying two items *)
3 4 5 [+] binary stack [9 3] equal.
```

Binary.



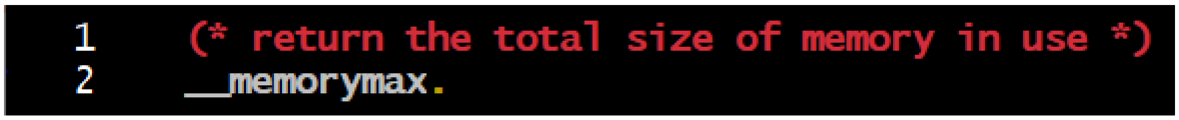
```
(* execute the app1 function *)
1 2 3 [+] app1 stack [5 1] equal.
```

App1.



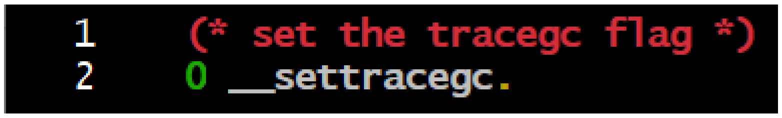
```
(* report the current size of memory allocation *)
__memoryindex.
```

__memoryindex.



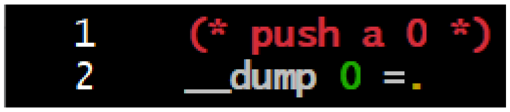
```
(* return the total size of memory in use *)
__memorymax.
```

__memorymax.



```
(* set the tracegc flag *)
0 __settracegc.
```

__settracegc.



```
(* push a 0 *)
__dump 0 =.
```

__dump.

Builtins

- ⟦choice ...⟧... B T F → ⟦...⟧... X
  If B is true, then X = T else X = F.
- ⟦setecho ...⟧... I → ⟦...⟧...
  [IMPURE] Sets value of echo flag for listing.
  I = 0: no echo, 1: echo, 2: with tab, 3: and linenumber.
- ⟦genrec ...⟧... [B] [T] [R1] [R2] → ⟦...⟧...
  Executes B, if that yields true, executes T.
  Else executes R1 and then [[[B] [T] [R1] R2] genrec] R2.
- ⟦construct ...⟧... [P] [[P1] [P2]] → ⟦...⟧... R1 R2
  Saves state of stack and then executes [P].
  Then executes each [Pi] to give Ri pushed onto saved stack.
- ⟦binary ...⟧... X Y [P] → ⟦...⟧... R
  Executes P, which leaves R on top of the stack.
  No matter how many parameters this consumes,
  exactly two are removed from the stack.
- ⟦app1 ...⟧... X [P] → ⟦...⟧... R
  Obsolescent. Executes P, pushes result R on stack.
- ⟦__memoryindex ...⟧... → ⟦...⟧... I
  [IMPURE] Pushes current value of memory.
- ⟦__memorymax ...⟧... → ⟦...⟧... I
  [IMPURE] Pushes value of total size of memory.
- ⟦__settracegc ...⟧... I → ⟦...⟧…
  [IMPURE] Sets value of flag for tracing garbage collection to I (= 0..6).
- ⟦__dump ...⟧... → ⟦...⟧… [..]
  debugging only: pushes the dump as a list.

Maintenance

Arity	Programmer says OK	Programmer says NOK
Checker says OK	OK	NOK
Checker says NOK	Unnecessarily slow	SLOW

This table shows what happens when the arity checker and the programmer have a different opinion about the arity of a condition. In Joy and joy1 the arity checker is not used, but in Moy and Foy it is.

Assuming that the programmer is right, the problem is in the upper right quadrant. When the programmer and the checker disagree, the checker wins. If the checker thinks that the arity is OK and it isn't, the program will fail with a runtime error, or worse, crash.

The second row is less of a problem. The program will work, but slower than necessary. Both the first row and the second row can be reported by the checker. The first row will be reported with `info:`; the second with `warning:`. This allows the programmer to make corrective actions such that the arity is OK.

Of course, the possibility remains that both the checker and the programmer are wrong. In any case, the use of the arity checker is essential in achieving good performance and adherence to the Joy standard.

ᛚ

Tutorial

```
(* test whether an error occurred in a file *)
"ferror.joy" "r" fopen ferror swap fclose false =.
```

Ferror.

```
(* flush the stdout stream *)
stdout fflush stdout =.
```

Fflush.

```
(* write a number of bytes in a file *)
"test" "w" fopen [34 65 66 67 34 10] fwrite fclose.
$ rm test
```

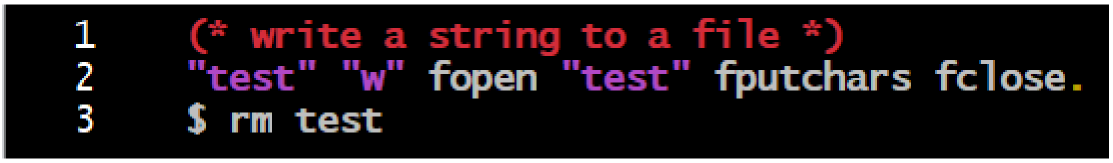
Fwrite.

```
(* write a factor to a file *)
"test" "w" fopen [1 2 3] fput fclose.
$ rm test
```

Fput.

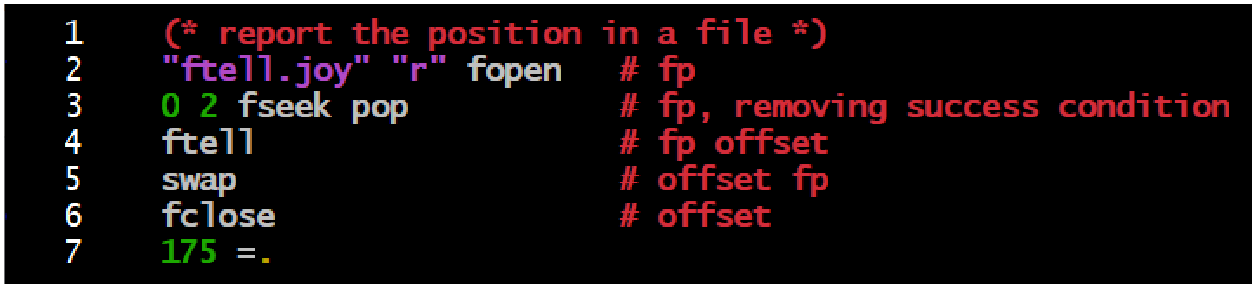
```
(* write a character to a file *)
"test" "w" fopen 39 fputch fclose.
$ rm test
```

Fputch.



```
(* write a string to a file *)
"test" "w" fopen "test" fputchars fclose.
$ rm test
```

Fputchars.



```
(* report the position in a file *)
"ftell.joy" "r" fopen   # fp
0 2 fseek pop           # fp, removing success condition
ftell                   # fp offset
swap                    # offset fp
fclose                  # offset
175 =.
```

Ftell.

```
(* set the seed of the pseudo random numbers *)
time srand.
```

Srand.

```
(* execute the app11 function *)
1 2 3 [+] app11 stack [5] equal.
```

App11.

```
(* X  Y  Z  [P]  app12 *)
1  2  3  4  [+]  app12 stack [6 5 1] equal.
```

App12.

Builtins

- ⟦ferror ...⟧... S → ⟦...⟧... S B
  [FOREIGN] B is the error status of stream S.
- ⟦fflush ...⟧... S → ⟦...⟧... S
  [FOREIGN] Flush stream S, forcing all buffered output to be written.
- ⟦fwrite ...⟧... S L → ⟦...⟧... S
  [FOREIGN] A list of integers are written as bytes to the current position of stream S.
- ⟦fput ...⟧... S X → ⟦...⟧... S
  [FOREIGN] Writes X to stream S, pops X off stack.
- ⟦fputch ...⟧... S C → ⟦...⟧... S
  [FOREIGN] The character C is written to the current position of stream S.
- ⟦fputchars ...⟧... S "abc..." → ⟦...⟧... S
  [FOREIGN] The string abc.. (no quotes) is written to the current position of stream S.
- ⟦ftell ...⟧... S → ⟦...⟧... S I
  [FOREIGN] I is the current position of stream S.
- ⟦srand ...⟧... I → ⟦...⟧...
  [IMPURE] Sets the random integer seed to integer I.
- ⟦app11 ...⟧... X Y [P] → ⟦...⟧... R
  Executes P, pushes result R on stack.
- ⟦app12 ...⟧... X Y1 Y2 [P] → ⟦...⟧... R1 R2
  Executes P twice, with Y1 and Y2, returns R1 and R2.

Maintenance

When all of the arities have been corrected, the code can be compiled.

The compiler simply dumps Joy source code. It does not do:

- compile time evaluation;
- inline body of definitions;
- inline quotations in combinators.

Each of these improvements might speedup the runtime evaluation, with some disadvantages:

- slower compilation;
- larger binaries created;
- more complicated design.

Simplicity is best, in this case. If an algorithm allows it, the compiled code can be linked with Roy and evaluated recursively; if not it can be linked with Soy without problem. Execution with Roy is faster. Translating the program to C is even faster, but where is the Joy of that?

comparison

A more elaborate comparison between compiling and interpreting is given below:

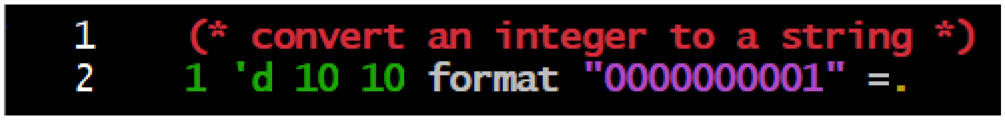
Advantages

- There is no symbol table. There is no need for a symbol table, because symbols have already been translated to addresses.
- Libraries need not be read. Definitions in libraries are not needed in the compiled executable. Those definitions that were used in the program have already been incorporated in that executable.
- There is only one binary file instead of a number of source files. This makes maintenance and deployment easier.
- Execution is always a bit faster than in the interpreter, even when exactly the same code is executed because the source code need not be read and also because the compiled code is more compact and cache friendly.
- The source code is not visible. The user of the program only gets to see the behaviour of the program and if that is satisfactory then there is no need to expose the source code.
- A programming language that comes with a compiler looks slightly more adult than a programming languages that only operates on source code.

Disadvantages

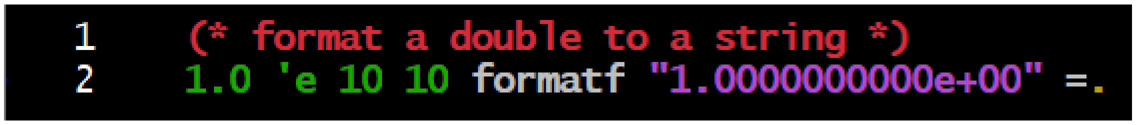
- A compiled program has limited functionality compared with the interpreter. It can do only one thing.
- Binaries take up more hard disk space than the source files that the interpreter uses.
- An edit-compile-run sequence is longer and slower than an edit-run sequence. This makes program development slower.
- There are not as many debugging options when running a compiled program as there are when running the interpreter.
- The arities must have been calculated in advance and must be correct. Unlike the interpreter that can handle wrong arities, although slower, the compiled code is meant to be fast and cannot be permissive in this respect.
- Not all of the programming language is supported. Builtins that use the symbol table are left out and there may be other discrepencies between the compiled code and the interpreter.

Tutorial



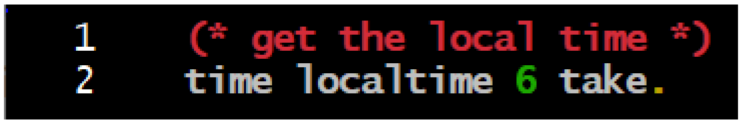
```
(* convert an integer to a string *)
1 'd 10 10 format "0000000001" =.
```

Format.



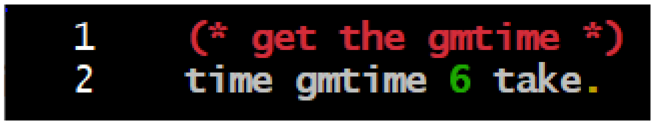
```
(* format a double to a string *)
1.0 'e 10 10 formatf "1.0000000000e+00" =.
```

Formatf.



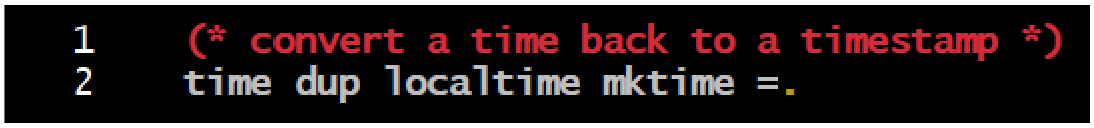
```
(* get the local time *)
time localtime 6 take.
```

Localtime.



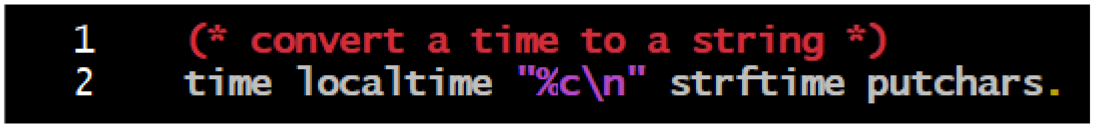
```
(* get the gmtime *)
time gmtime 6 take.
```

Gmtime.



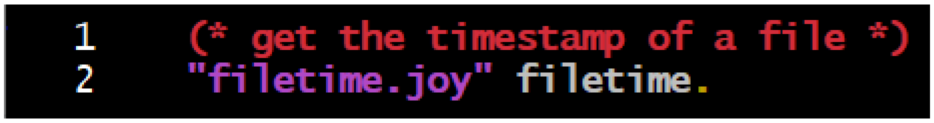
```
(* convert a time back to a timestamp *)
time dup localtime mktime =.
```

Mktime.



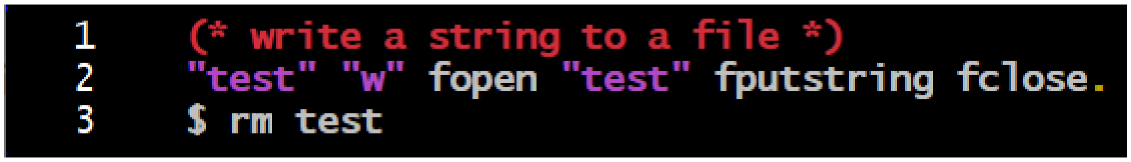
```
(* convert a time to a string *)
time localtime "%c\n" strftime putchars.
```

Strftime.



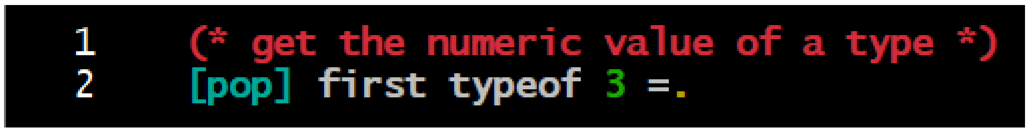
```
(* get the timestamp of a file *)
"filetime.joy" filetime.
```

Filetime.



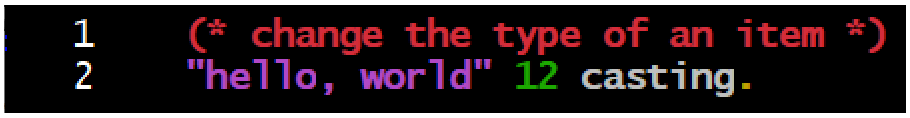
```
(* write a string to a file *)
"test" "w" fopen "test" fputstring fclose.
$ rm test
```

Fputstring.



```
(* get the numeric value of a type *)
[pop] first typeof 3 =.
```

Typeof.



```
(* change the type of an item *)
"hello, world" 12 casting.
```

Casting.

Builtins

- ⟦format ...⟧... N C I J → ⟦...⟧... S
  S is the formatted version of N in mode C
  ('d or 'i = decimal, 'o = octal, 'x or
  'X = hex with lower or upper case letters)
  with maximum width I and minimum width J.

- ⟦formatf ...⟧... F C I J → ⟦...⟧... S
  S is the formatted version of F in mode C
  ('e or 'E = exponential, 'f = fractional,
  'g or G = general with lower or upper case letters)
  with maximum width I and precision J.

- ⟦localtime ...⟧... I → ⟦...⟧... T
  Converts a time I into a list T representing local time:
  [year month day hour minute second isdst yearday weekday].
  Month is 1 = January ... 12 = December;
  isdst is a Boolean flagging daylight savings/summer time;
  weekday is 1 = Monday ... 7 = Sunday.

- ⟦gmtime ...⟧... I → ⟦...⟧... T
  Converts a time I into a list T representing universal time:
  [year month day hour minute second isdst yearday weekday].
  Month is 1 = January ... 12 = December;
  isdst is false; weekday is 1 = Monday ... 7 = Sunday.

- ⟦mktime ...⟧... T → ⟦...⟧... I
  Converts a list T representing local time into a time I.
  T is in the format generated by localtime.

- ⟦strftime ...⟧... T S1 → ⟦...⟧... S2
  Formats a list T in the format of localtime or gmtime
  using string S1 and pushes the result S2.

- ⟦filetime ...⟧... F → ⟦...⟧... T
  [FOREIGN] T is the modification time of file F.

- ⟦fputstring ...⟧... S “abc...” → ⟦...⟧... S
  [FOREIGN] == fputchars, as a temporary alternative.

- ⟦typeof ...⟧... X → ⟦...⟧... I
  [EXT] Replace X by its type.
- ⟦casting ...⟧... X Y → ⟦...⟧... Z
  [EXT] Z takes the value from X and uses the value from Y as its type.

Maintenance

Some bugs have been removed from Joy. They are still present in the legacy version. Not every correction is mentioned:

linenum

This behaviour was promised in `j09imp.html`, but is not present in the legacy version. The input stack needs to remember not only the file pointer, but also the linenumber, such that this linenumber can be continued after an included file was processed. As it happens, the filename is also remembered, and is currently also used in error messages.

escape

Character escape sequences are easier to remember when this is enabled: all ASCII values between 8 and 13 inclusive can then be escaped in a symbolic way. Not that there is an urgent need for this addition, because escaping can also be done numerically: `'\n` is equal to `'\010`.

numbers

The document `j09imp.html` promises that octal and hexadecimal numbers are supported, and indeed those are supported by `strtoll`. The old code, however, reads digits, +, -, ., E, and e. The reading should stop as soon as a number has been read. More specifically, if a number is followed by a stop and then a non-digit character, the non-digit character and the stop should be pushed back into the input stream.

hexadecimals

As mentioned in the previous paragraph, hexadecimal digits A-F, or a-f were not considered as part of a number. This has been corrected.

octals

As was mentioned octal numbers were supported. What that means is that as soon as an octal number has been spotted, it is reported as such. More specifically: `08` is parsed as two numbers, `0` and `8`. The reason that it cannot be an octal number is that `8` is not an octal digit. Spaces between tokens are only mandatory when needed to separate two tokens. Even more: `00` can be parsed as two numbers, both 0. That is not what is done in Joy (it is that way in Moy).

floats

`1.` is a valid floating point in C, but Joy can have its own definition of a floating point. In the

Joy definition it is required to have digits on both sides of the decimal point. So, `1. ` with a non-digit character after the decimal point is parsed as an integer, followed by a full stop, followed by whatever comes thereafter. Joy is somewhat tied to C as the implementation language but need not follow every quirck of that language.

strings

Ideally, the output from writefactor should be valid Joy source code. The old version prints a string as `"%s"` and that fails if the string contains a `"` or a newline. Also, if there are unreadable characters in the string, the output will look strange. This has been corrected.

displaymax

The legacy version does not check overflow of the display of local symbols. This has been corrected. The displaymax is used for both modules and local symbols.

checkstack

There is the problem of the program `[1 2 3] [pop] map.` The builtin `map` needs an entry on the stack that can be used in the newly created list. The `pop` makes sure that such entry does not exist. This causes a crash. This has been corrected. The same problem also occurs in many other builtins and may not have been corrected everywhere.

compare

It would be good to have only one definition of equality. `Compare` compares each type with every other type and is a robust way to enforce the same kind of equality in `compare,equal,case, in,has,=,<`, and other comparison operators. But `in` has a problem: it breaks `grmtst.joy`. This problem was finally mitigated by using a nickname: plus and ast instead of + and *.

intern

The builtin `intern` allows interning of symbols with spaces or other characters that do not adhere to the naming restrictions of identifiers. This looks like a mistake, but it is not necessary to have this corrected. After all, there exists a different way of accessing symbols: `"symbol" intern == [symbol] first`. In both cases, the symbol `symbol` will be placed on the stack. There are 3 exceptions to the equality of `intern` and `[ ... ] first`: `false`, `true`, and `maxint`. Thus: `"false" intern != [false] first`. The word `intern` returns the function `false`, whereas the construct with `first` returns the value `false`.

getenv

The function `getenv` can return a NULL pointer and that should be replaced by "", an empty string. A NULL pointer is not a valid string in Joy. If it would be accepted as such, all locations where a string is used, need protection against NULL pointers. It is better to tackle the problem at the source and replace the NULL with an empty string.

helpdetail

The function `helpdetail` could be improved. In case of the values `false`, `true`, and `maxint`, it is desirable to print the description of the functions, not the description of the values. Thus some translation from value to function needs to take place. This has been corrected.

getch

There are a number of file operations that send data to output, for example `putch`. So, why not have the same number of operations on input? `getch` fills that gap. This `getch` is only available in 42minjoy. This small version of Joy has only `getch` and `putch` to do input and output.

sametype

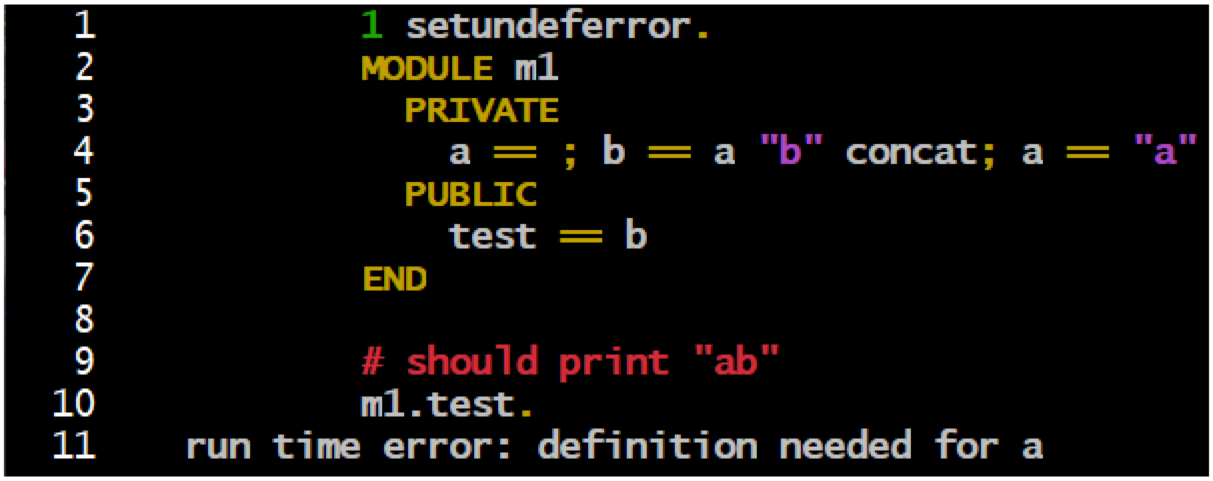
The predicate `sametype` was lost during the development of Joy and is a useful addition. It allows some datatype specific predicates to be replaced by `sametype`, if needed.

not

It looks ok to have `not` only available for BOOLEAN_ and SET_. After all, `not` inverts a value and while it is possible to change every other value into 0, inverting 0 is only possible if there is only 1 value to invert to. Also, if `not` is available for other datatypes, it overlaps functionality with `null`.

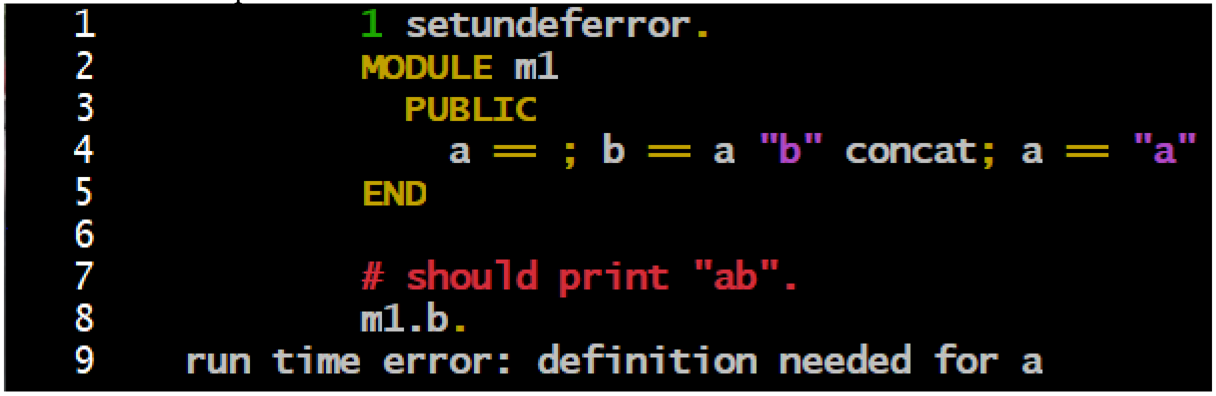
modules

Manfred: “My implementation of HIDE contains one error,” also present in modules:



```
        1 setundeferror.
        MODULE m1
          PRIVATE
            a == ; b == a "b" concat; a == "a"
          PUBLIC
            test == b
        END

        # should print "ab"
        m1.test.
run time error: definition needed for a
```

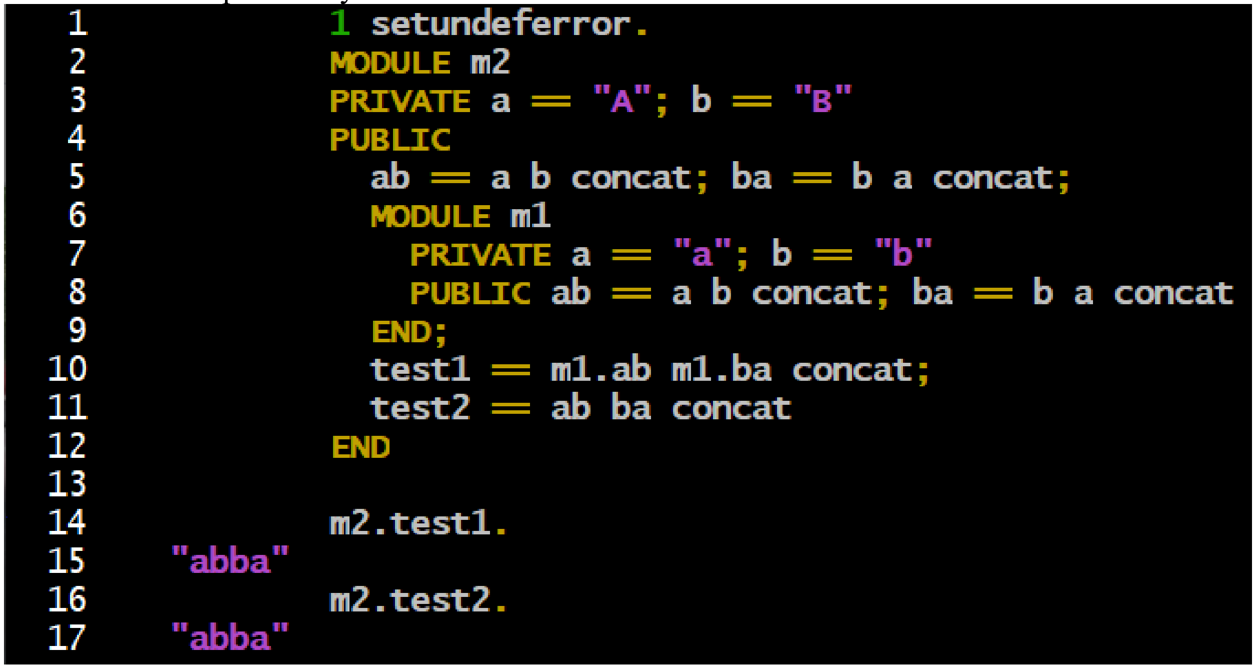
Local definitions cannot call each other. The empty definition of “a” does not help.
This feature is also present in Public definitions:



```
        1 setundeferror.
        MODULE m1
          PUBLIC
            a == ; b == a "b" concat; a == "a"
        END

        # should print "ab".
        m1.b.
run time error: definition needed for a
```

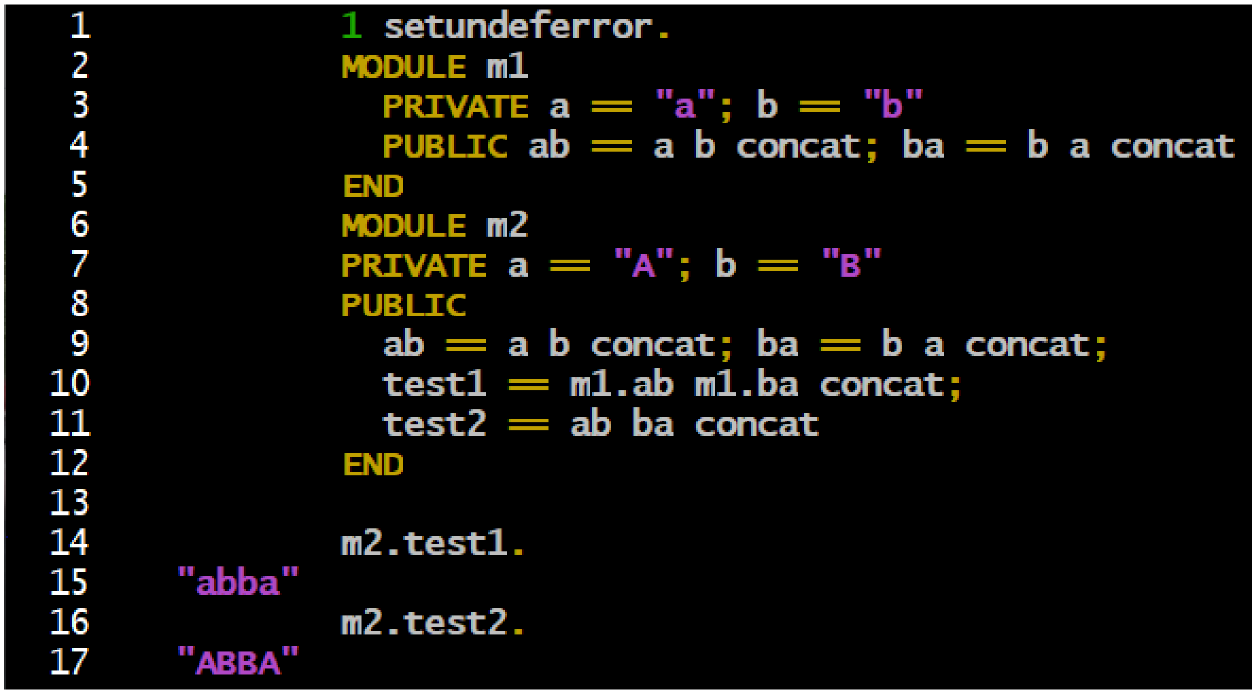
This limitation makes modules less attractive compared to global definitions. Unlike global definitions they cannot be mutually recursive.

There is also the possibility to define modules within modules:



```
        1 setundeferror.
        MODULE m2
        PRIVATE a == "A"; b == "B"
        PUBLIC
          ab == a b concat; ba == b a concat;
          MODULE m1
            PRIVATE a == "a"; b == "b"
            PUBLIC ab == a b concat; ba == b a concat
          END;
          test1 == m1.ab m1.ba concat;
          test2 == ab ba concat
        END

        m2.test1.
"abba"
        m2.test2.
"abba"
```

The m2.test2 should print “ABBA.” It looks like the “ab” in m1 overrides the “ab” in “m2.”
If module “m1” is specified before “m2” everything is fine:



```
    1 setundeferror.
    MODULE m1
     PRIVATE a == "a"; b == "b"
     PUBLIC ab == a b concat; ba == b a concat
    END
    MODULE m2
    PRIVATE a == "A"; b == "B"
    PUBLIC
     ab == a b concat; ba == b a concat;
     test1 == m1.ab m1.ba concat;
     test2 == ab ba concat
    END

    m2.test1.
"abba"
    m2.test2.
"ABBA"
```

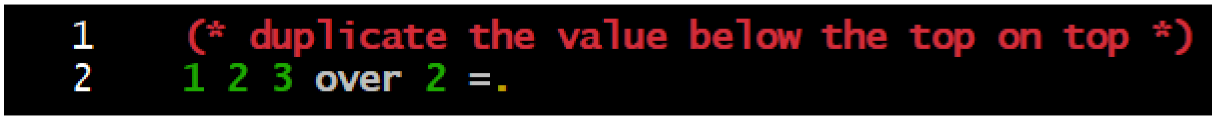
It can be concluded that modules-within-modules are not properly supported in the Legacy version. It should be noted that this feature is not mentioned in the user manual and also not demonstrated in the test file “modtst.joy.”

All of this has been corrected in the current version of Joy. The proper way to support this is to read the sections between HIDE .. END and MODULE .. END twice. During the first read all defined symbols are entered in the symbol table, making them available during the second read. Actually reading source code more than once is not really possible, so the first read builds a list of tokens that can be used in the second read.

Definitions within modules are stored in the symbol table together with the module name. Local definitions are stored with a numeric module name. These qualified names are used during a first lookup in the hash table. If that fails, a secondary lookup without the qualifications is tried. All of this logic is not simple, but it looks like it satisfies the requirements.

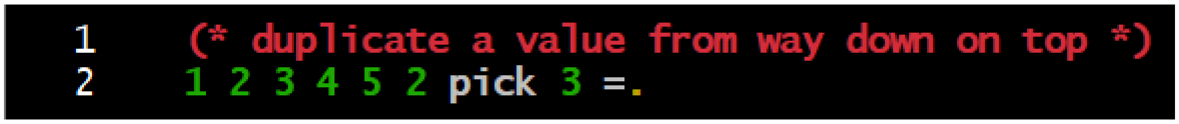
ᛞ

Tutorial



```
(* duplicate the value below the top on top *)
1 2 3 over 2 =.
```

Over.



```
(* duplicate a value from way down on top *)
1 2 3 4 5 2 pick 3 =.
```

Pick.

Builtins

- ⟦over ...⟧... X Y → ⟦...⟧... X Y X
  [EXT] Pushes an extra copy of the second item X on top of the stack.
- ⟦pick ...⟧... X Y Z 2 → ⟦...⟧... X Y Z X
  [EXT] Pushes an extra copy of nth (e.g. 2) item X on top of the stack.

Maintenance

Joy can be compared with the Lambda Calculus:

variable ::= ‘v’ | variable ‘′’

λ-term ::= variable | ‘(’ λ-term$_1$ λ-term$_2$ ‘)’ | ‘(’ λ variable λ-term ‘)’

Joy does not have variables, instead it has constants. And because there are no variables, there is also no lambda abstraction.

constant ::= swap | dup | pop …

λ-term ::= constant | ‘(’ λ-term$_1$ λ-term$_2$ ‘)’

So far, Joy does not differ from the 100 year old combinatory calculus, that also uses constants, at least S and K. But here the languages start to diverge:

In the combinatory calculus λ-term$_1$ is a function that takes λ-term$_2$ as parameter. In Joy both are functions that are evaluated in sequential order: first λ-term$_1$, then λ-term$_2$. This order makes the parentheses unnecessary. So, here is the full grammar of Joy:

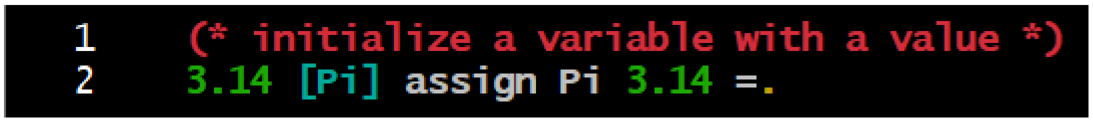
constant ::= swap | dup | pop …

term ::= /* empty */ | term constant

This simplicity comes with a price: the explicit shuffling that is needed to get parameters into the correct location, as done by the constants “swap”, “dup”, “pop.” Languages that have named parameters can simply use those names and have no need of these shuffling operators.

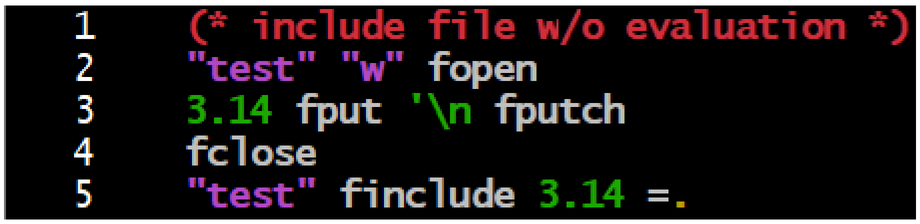
ᛟ

Tutorial



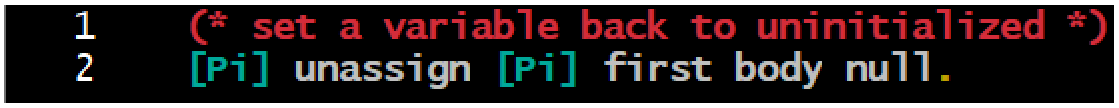
```
(* initialize a variable with a value *)
3.14 [Pi] assign Pi 3.14 =.
```

Assign.



```
(* include file w/o evaluation *)
"test" "w" fopen
3.14 fput '\n fputch
fclose
"test" finclude 3.14 =.
```

Finclude.



```
(* set a variable back to uninitialized *)
[Pi] unassign [Pi] first body null.
```

Unassign.

Builtins

- ⟦assign ...⟧... V [N] → ⟦...⟧...
  [IMPURE] Assigns value V to the variable with name N.
- ⟦finclude ...⟧... S → ⟦...⟧...
  [Foreign] Reads Joy source code from file S and pushes it onto stack.
- ⟦unassign ...⟧... [N] → ⟦...⟧...
  [IMPURE] Sets the body of the name N to uninitialized.

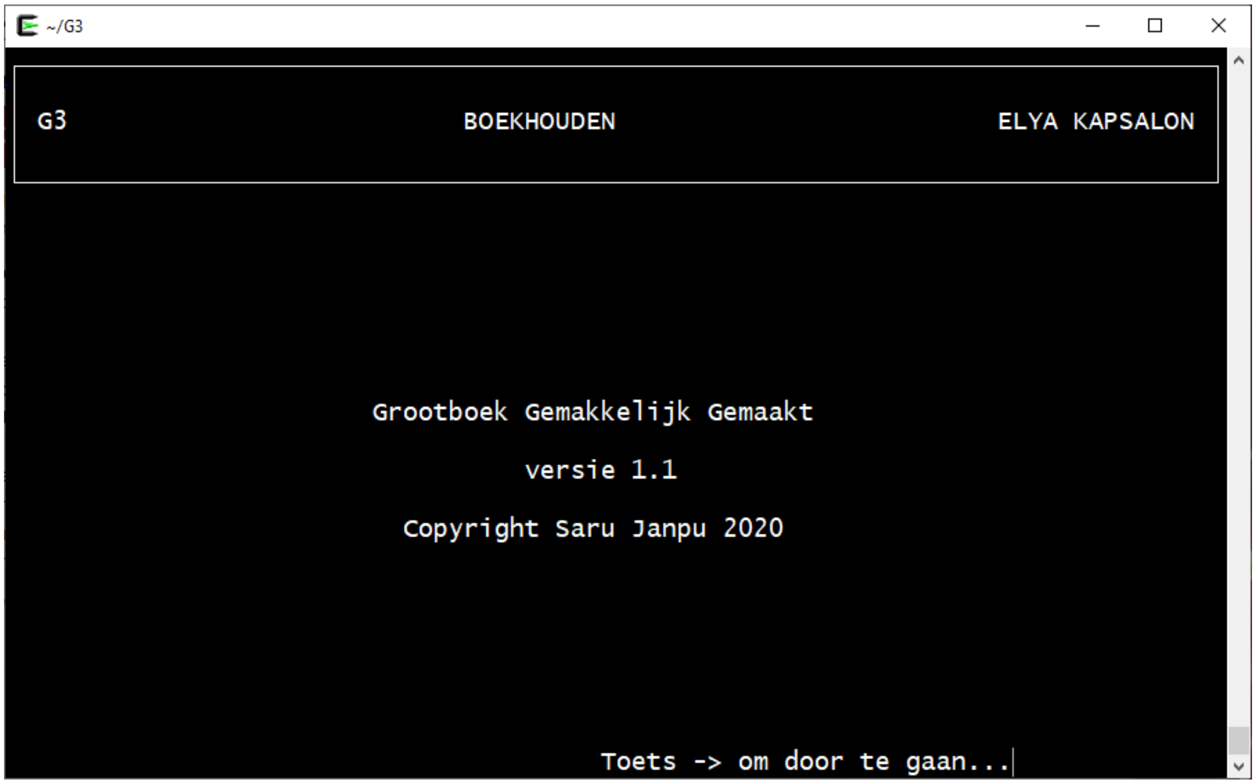
Maintenance

Assignment ruins everything. The syntax is taken from chapter 18 in Symbolic Processing in Pascal. If variables are assigned only once, it can still be seen as functional. The variables will then produce the same value, everytime they are used.

But of course, once variables are introduced, and they are also immediately global variables, they will be used and the language becomes an imperative language. Nothing wrong with that, but it is not according to design.

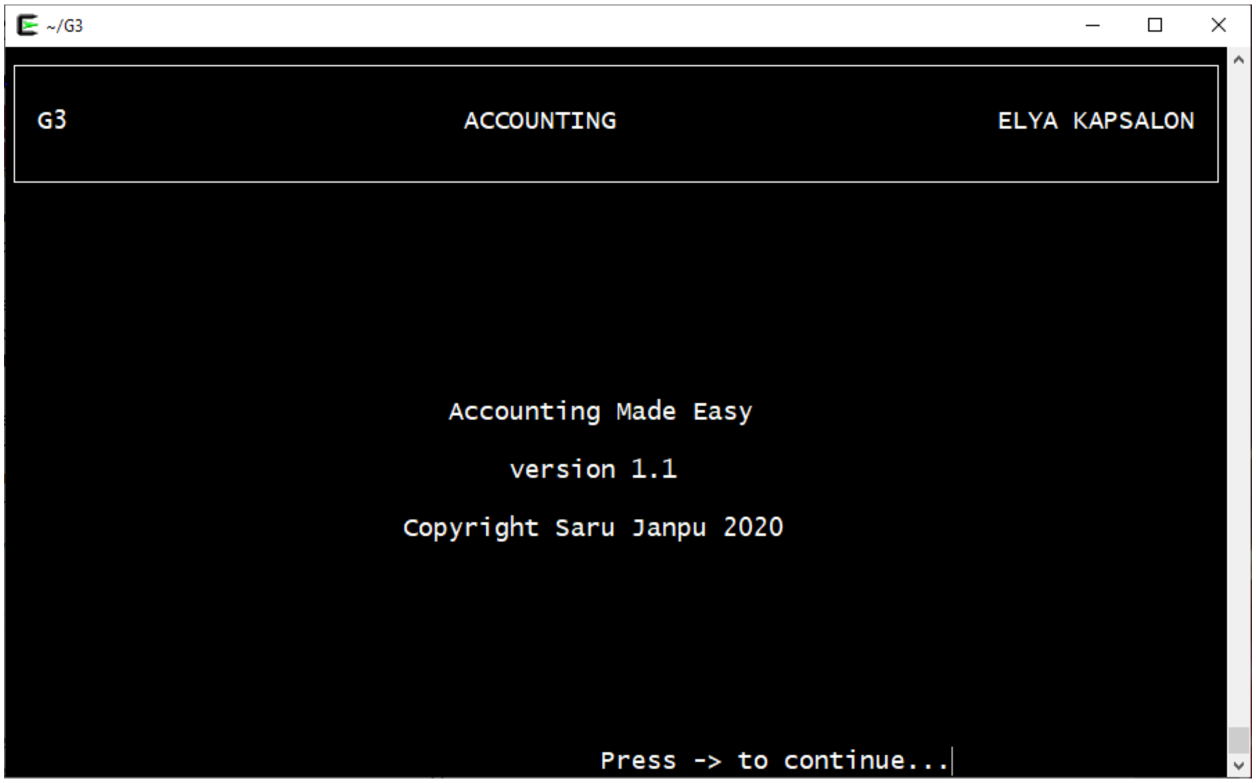
Even so, it has to be admitted that the quadratic formula is more readable with names than it is without. It is still postfix and some users don’t like that, because their natural language is also not postfix.

A



```
~/G3

 G3                          BOEKHOUDEN                        ELYA KAPSALON



                      Grootboek Gemakkelijk Gemaakt

                              versie 1.1

                       Copyright Saru Janpu 2020



                                        Toets -> om door te gaan...
```

This is the start screen of G3, an application written in Joy, in Dutch translation in order to show where the name G3 comes from. The English translation will be given on the next page. In the upper right is the company that uses this accounting program and the Copyright line mentions the issuing company. Both companies are real companies, registered at the Chamber of Commerce. In the upper left is the name of the program. The middle of the header mentions what this screen is about. In the lower right there is a clue about what keys can be pressed. There are only two such clues in the program: how to continue and how to leave. The bottom left is reserved for error messages. There are currently no error messages.



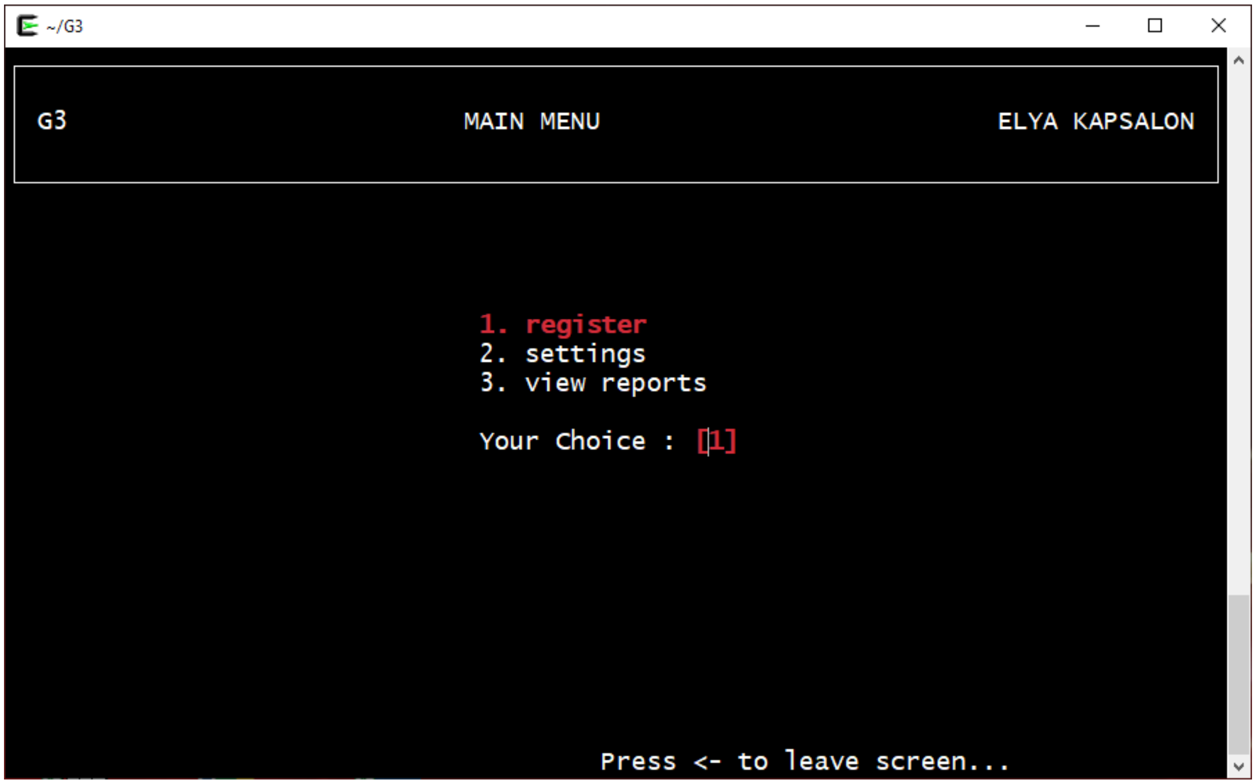
```
G3                    ACCOUNTING                    ELYA KAPSALON




                  Accounting Made Easy

                      version 1.1

               Copyright Saru Janpu 2020




                             Press -> to continue...
```

The program starts by reading “lang.joy” that contains texts that have been translated. One text that must always be customized is the name of the company in the upper right part of the screen.

When using a new program there are two obstacles: how to start the program and how to end it. Starting the program must have been solved, when this screen appears, so only ending the program remains. That question will be answered in the next chapter. In this screen only the right arrow key is expected.

This screen may look like something from the eighties. Indeed, it comes from the eighties. The Copyright statement showed a different, now defunct, company. Some of the look-and-feel is copied from the original. Architectural and design decisions that were necessary at the time are no longer necessary. What is left is a program that reads from the keyboard and displays characters to the screen. What more can be expected from an accounting program that uses texts and numbers?

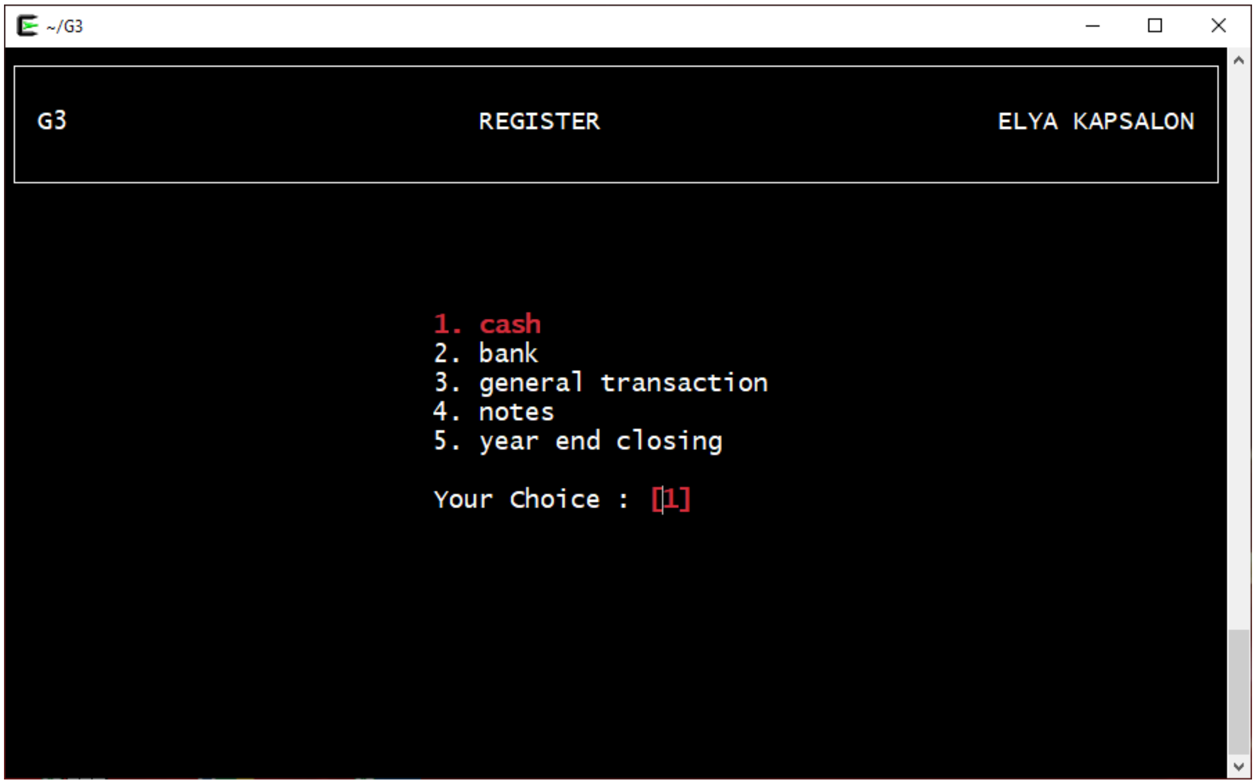
B



```
~/G3

 G3                          MAIN MENU                          ELYA KAPSALON


                          1. register
                          2. settings
                          3. view reports

                          Your Choice : [1]



                                 Press <- to leave screen...
```

This is the main menu that also answers the question how the application can be ended. This can be done with the left arrow key. In fact, all navigation in the menu's and screens is done with the help of the cursor keys. In addition to that, menu choices can also be made by typing the digit in front of the menu line~~, or the first character of the description~~. That is why the 3rd description is "view reports" and not "reports."

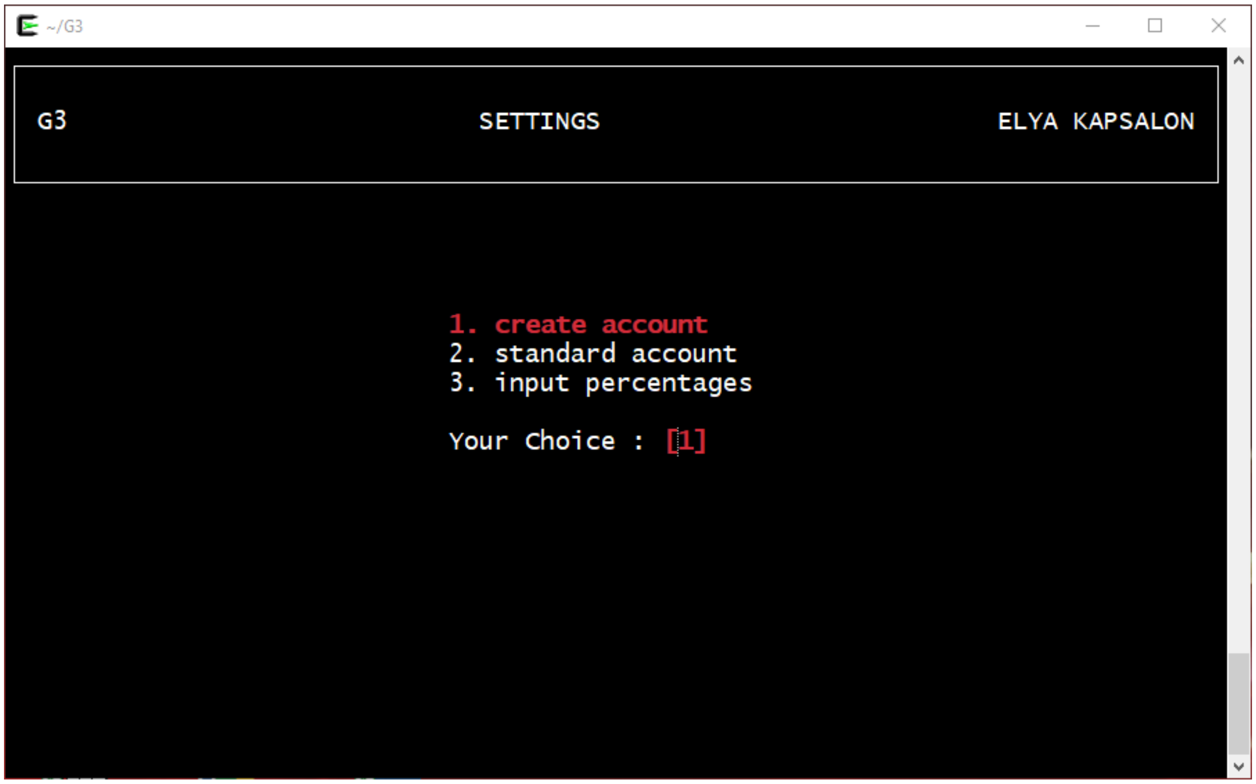
C



```
~/G3

G3                          REGISTER                          ELYA KAPSALON

                         1. cash
                         2. bank
                         3. general transaction
                         4. notes
                         5. year end closing

                         Your Choice : [1]
```

This is the menu that allows postings to be made. Cash and Bank allow postings with Cash or Bank as one of the accounts; General Transaction allows a transfer of money from one account to another. Notes allows adding references to an existing posting and Year End Closing creates a post that reverses all Profit & Loss accounts, moving the result to a result account. This is part of the Year End Closing Procedure.

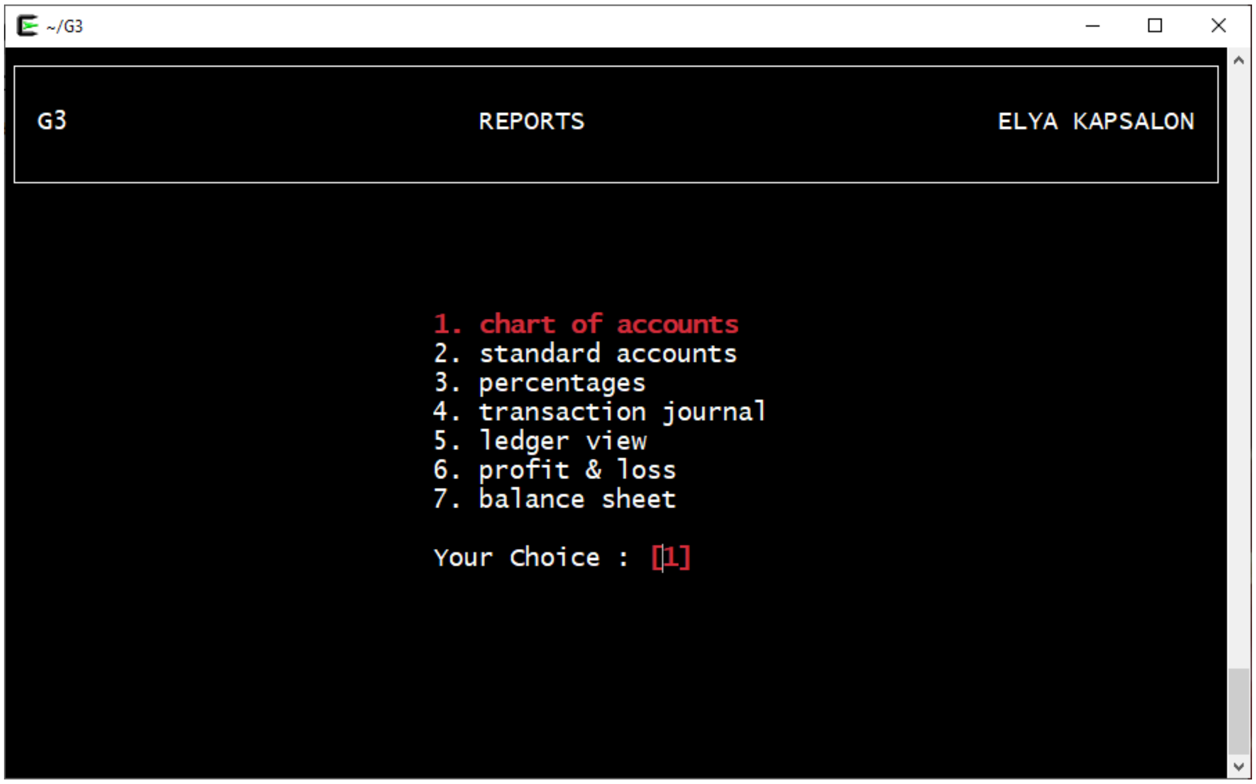
D



```
~/G3

G3                         SETTINGS                    ELYA KAPSALON

                    1. create account
                    2. standard account
                    3. input percentages

                    Your Choice : [1]
```

The settings menu comes second in the main menu. It should be customized before starting to make postings. As it happens, posting is possible even before these settings have been filled. The first entry allows creating accounts, the second connects Cash and Bank with an account and the second and third can be used to connect VAT codes to an account and a percentage. The automatic VAT calculation makes posting a little easier.

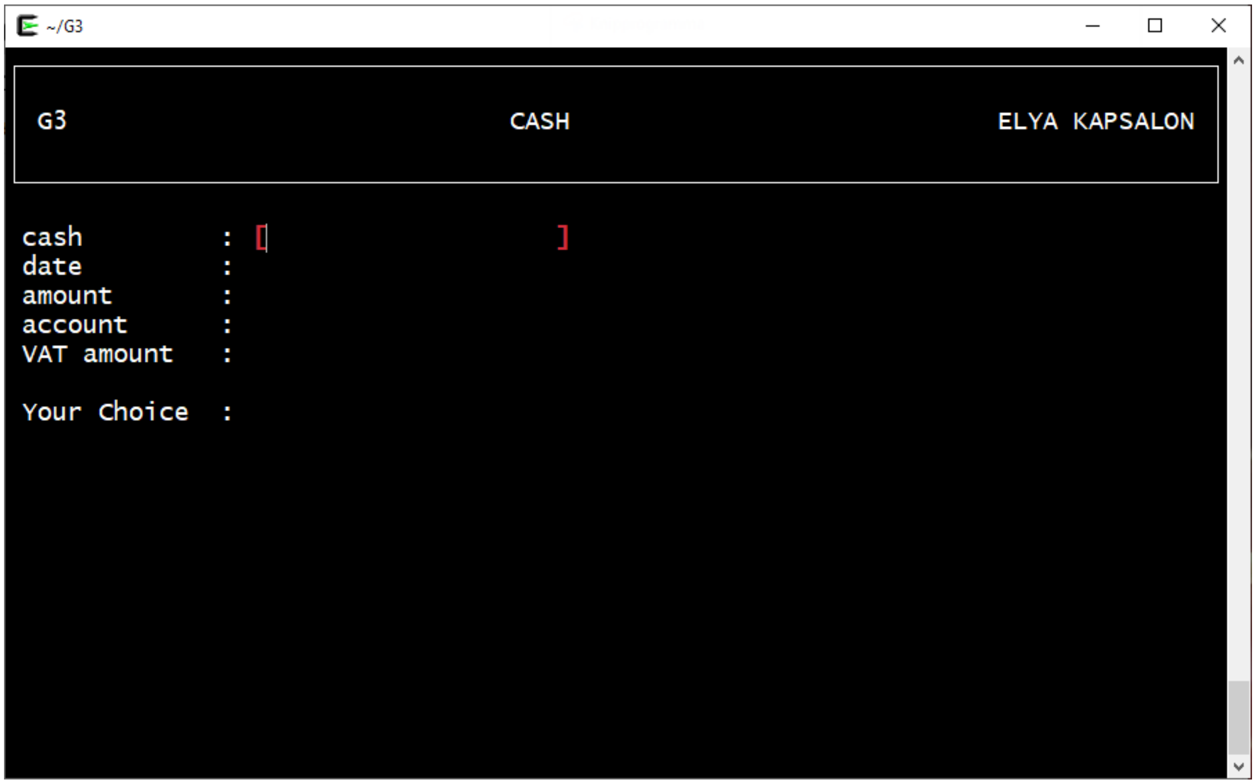
E



```
~/G3

G3                          REPORTS                          ELYA KAPSALON


                       1. chart of accounts
                       2. standard accounts
                       3. percentages
                       4. transaction journal
                       5. ledger view
                       6. profit & loss
                       7. balance sheet

                       Your Choice : [1]
```

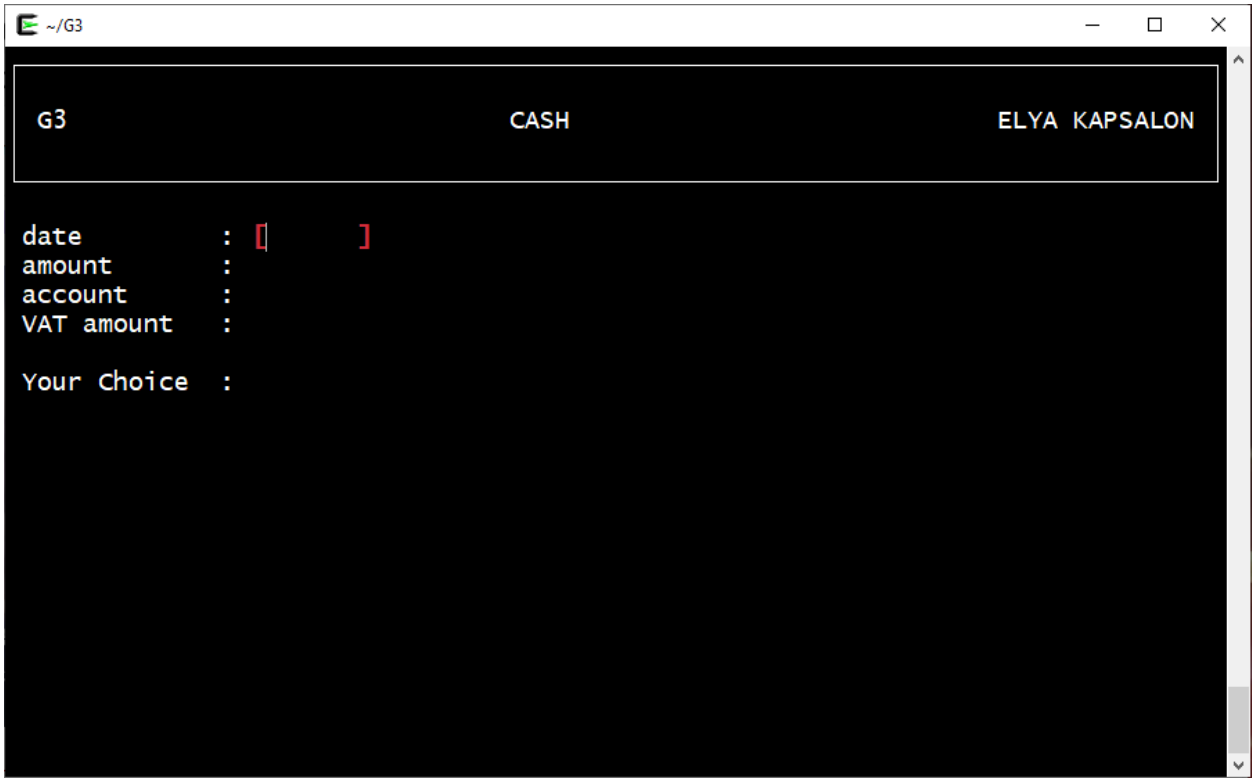
The reports menu comes third in the main menu. The first three entries allow an overview of the standard settings. The Transaction Journal allows an overview of all postings made. The Ledger View restricts the view of postings to one account and also presents a summary. The Profit & Loss and Balance Sheet present the standard financial overviews and should agree about the amount of Net Profit or Net Loss.

F



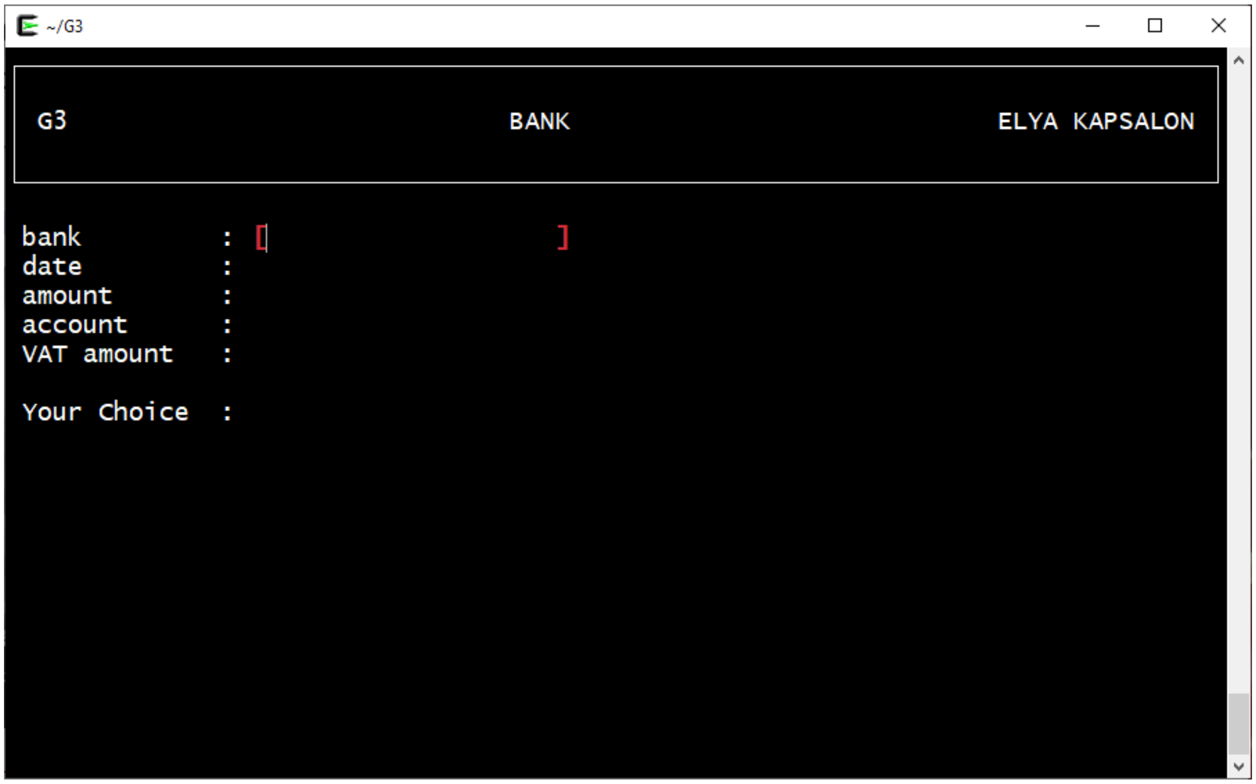
```
G3                              CASH                         ELYA KAPSALON

cash          : [                   ]
date          :
amount        :
account       :
VAT amount    :

Your Choice   :
```

This is the initial screen that allows Cash payments to be registered. As the standard Cash account may not have been customized yet, the system asks for it in the first line. As soon as the connection has been established, this first line is not presented again. Dates need to be entered as ddmmyy. Accounts can be used before they have been entered in the settings menu. All fields, except the VAT amount are mandatory. The system has no knowledge about the real world and cannot error when a wrong date, amount, or account is entered. The Your Choice field allows posting, leaving the screen without posting, or start editing the fields on the screen. No error messages are given.


```
~/G3

 G3                              CASH                          ELYA KAPSALON

date          : [      ]
amount        :
account       :
VAT amount    :

Your Choice   :
```

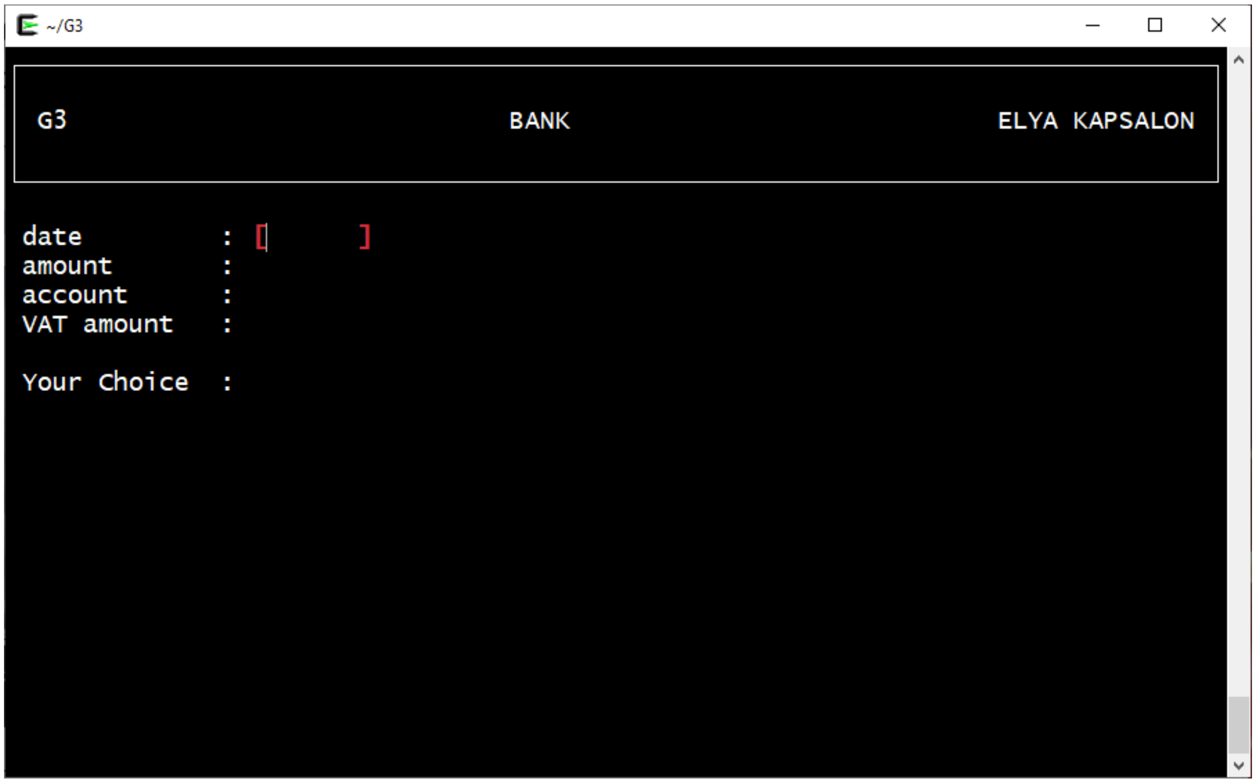
This is what the Cash screen looks like after the establishment of the standard Cash account.

G



```
~/G3

G3                          BANK                         ELYA KAPSALON

bank         : [                   ]
date         :
amount       :
account      :
VAT amount   :

Your Choice  :
```

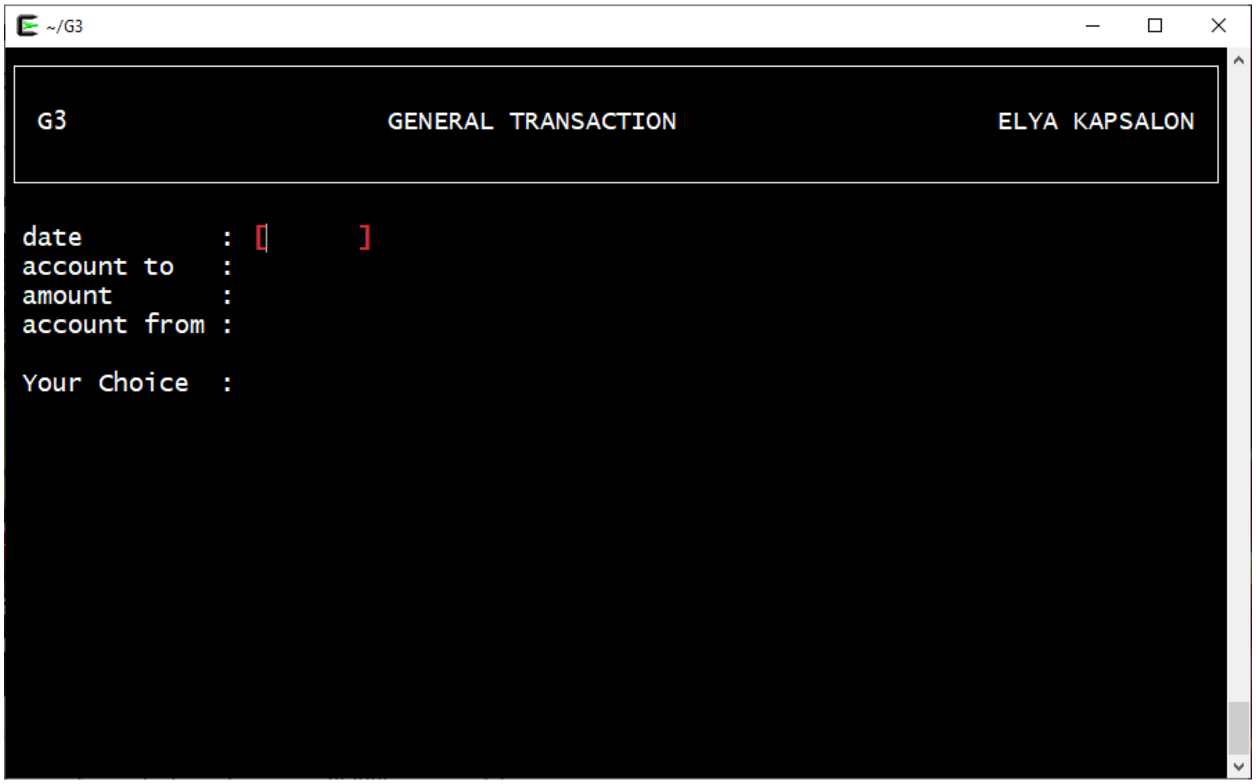
The Bank initial screen is similar to the Cash initial screen. Payments by Bank are usually recurring payments. The accounts can be as large as 19 digits and it may be tempting to use the bank account number itself as account number. But IBAN numbers have characters and accounts can only consist of digits, so that is not possible.



```
~/G3

G3                              BANK                          ELYA KAPSALON

date          : [      ]
amount        :
account       :
VAT amount    :

Your Choice   :
```

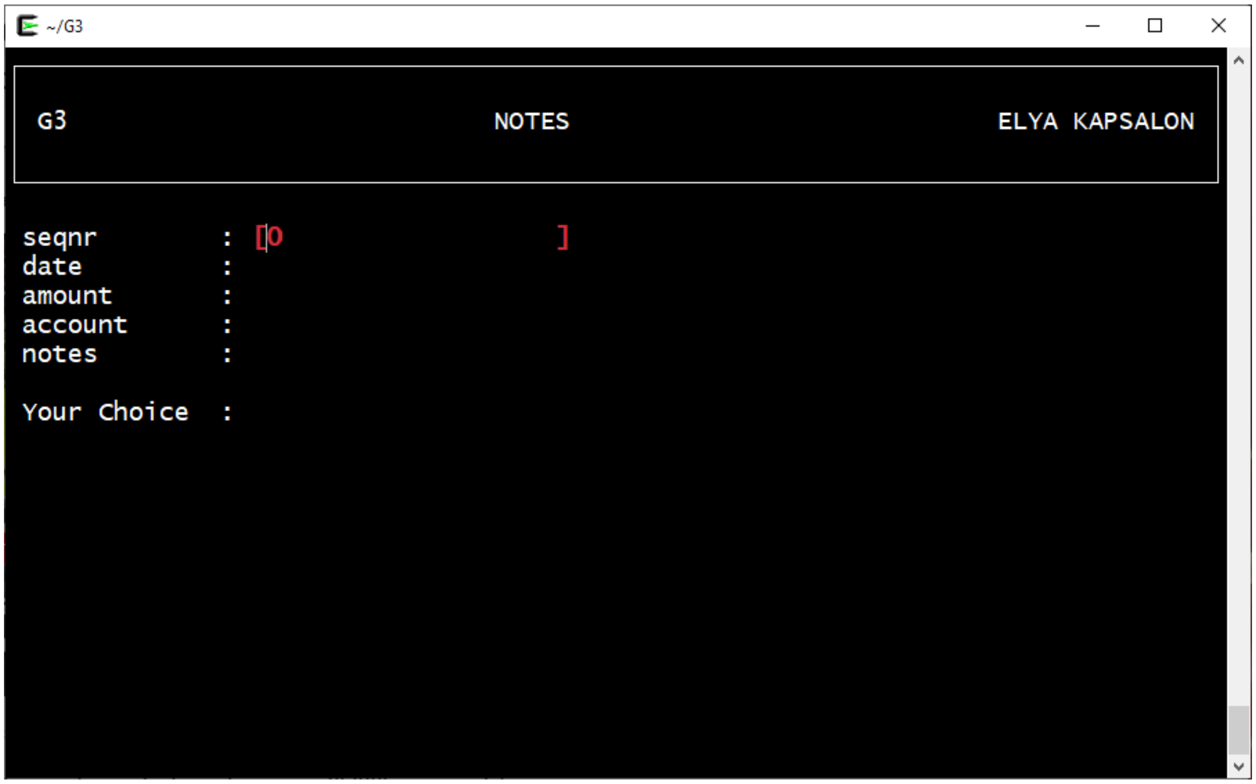
This is what the Bank screen looks like after establishing the connection with the standard account.

H



```
~/G3

G3                      GENERAL TRANSACTION                    ELYA KAPSALON

date         : [      ]
account to   :
amount       :
account from :

Your Choice  :
```

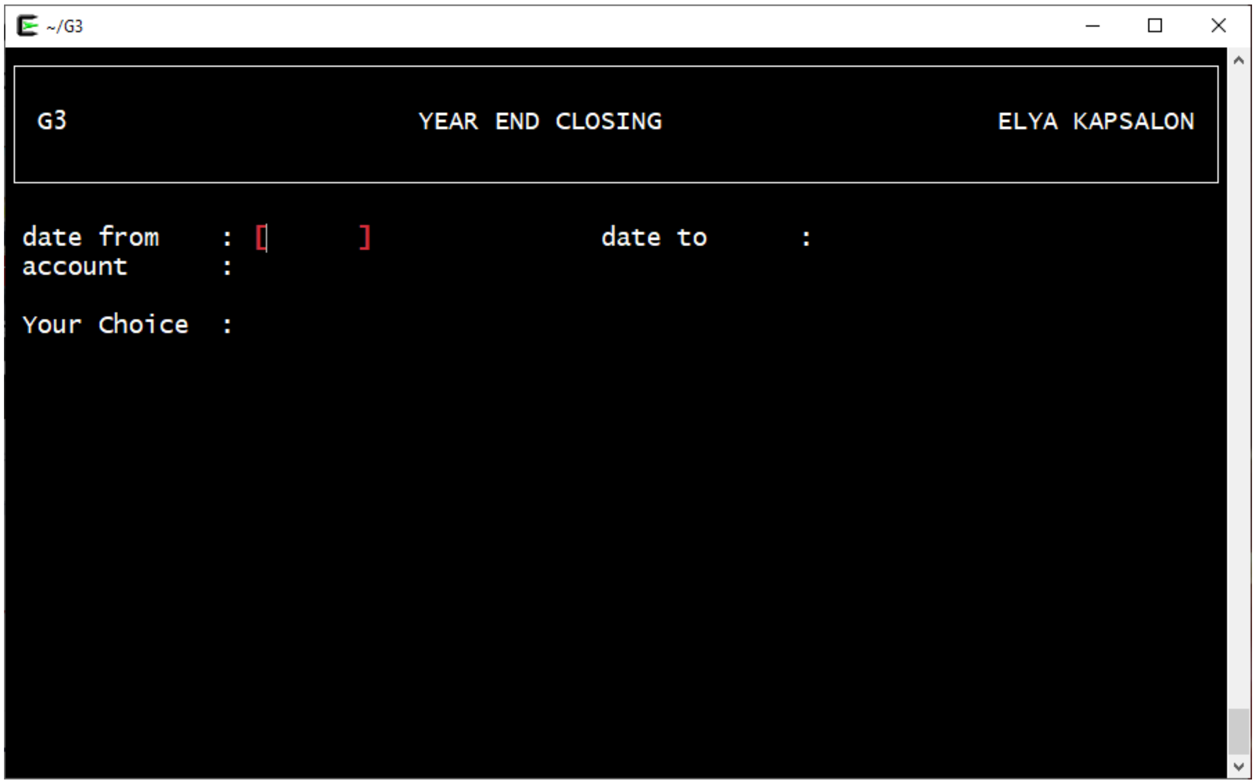
This is the general accounting data entry screen. It allows registration of a transfer of money from one account to another. Date comes first and should be the date when the financial event occurred. The “account to” field is the receiving account of the amount that is entered in the next field. The “account from” field receives the negated amount. If the account already exists, the description is printed on the same line. In this screen there is no automatic VAT calculation.



```
~/G3

 G3                          NOTES                          ELYA KAPSALON

seqnr          : [0                  ]
date           :
amount         :
account        :
notes          :

Your Choice    :
```

This is the notes screen. It can be used to establish a connection between a posting and the financial event in the real world. Accounts are virtual by definition, but should reflect the financial events that occurred. This screen allows adding a note to an existing posting. The first line of the posting is displayed. Only the notes are stored and they are stored in a separate file.

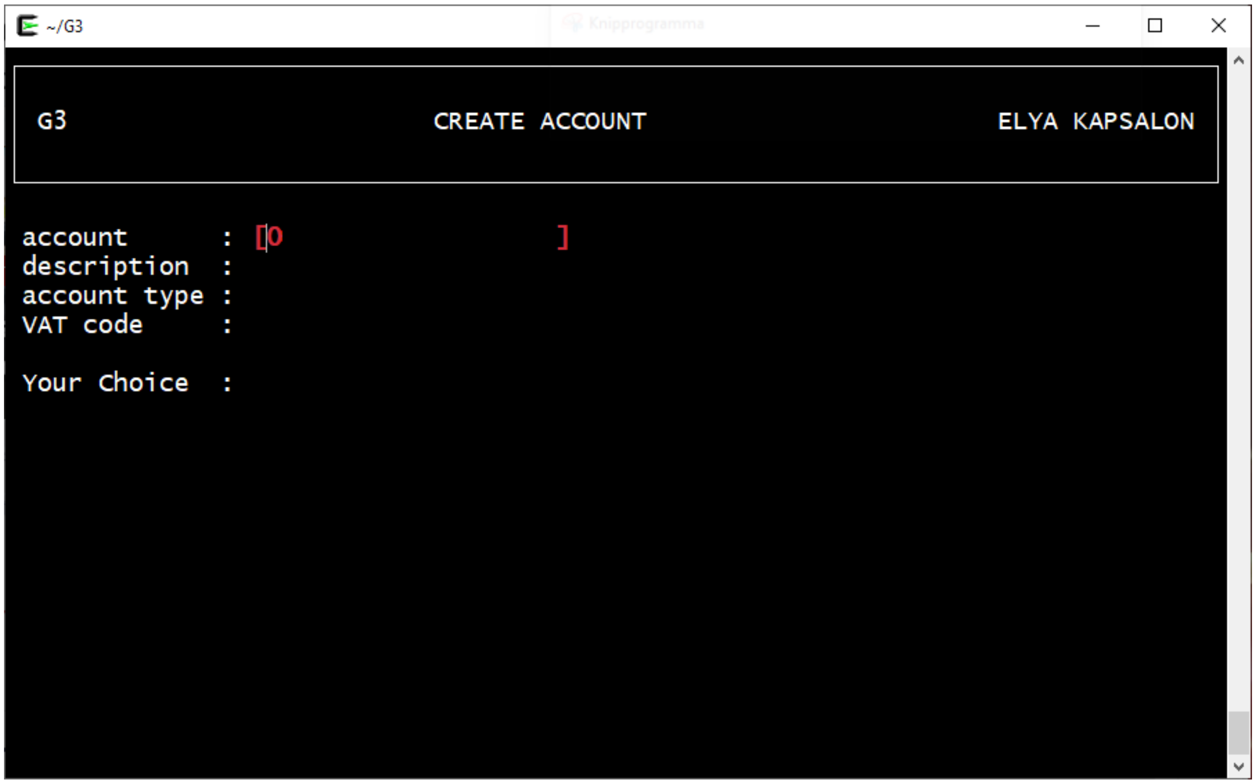
J



```
~/G3

 G3                        YEAR END CLOSING                      ELYA KAPSALON

date from     : [       ]                date to      :
account       :

Your Choice   :
```

The Year End Closing allows a posting to be made that reverses all Profit & Loss accounts and stores the result in the account given. All fields in this screen are mandatory. The dates need not cover a complete year.

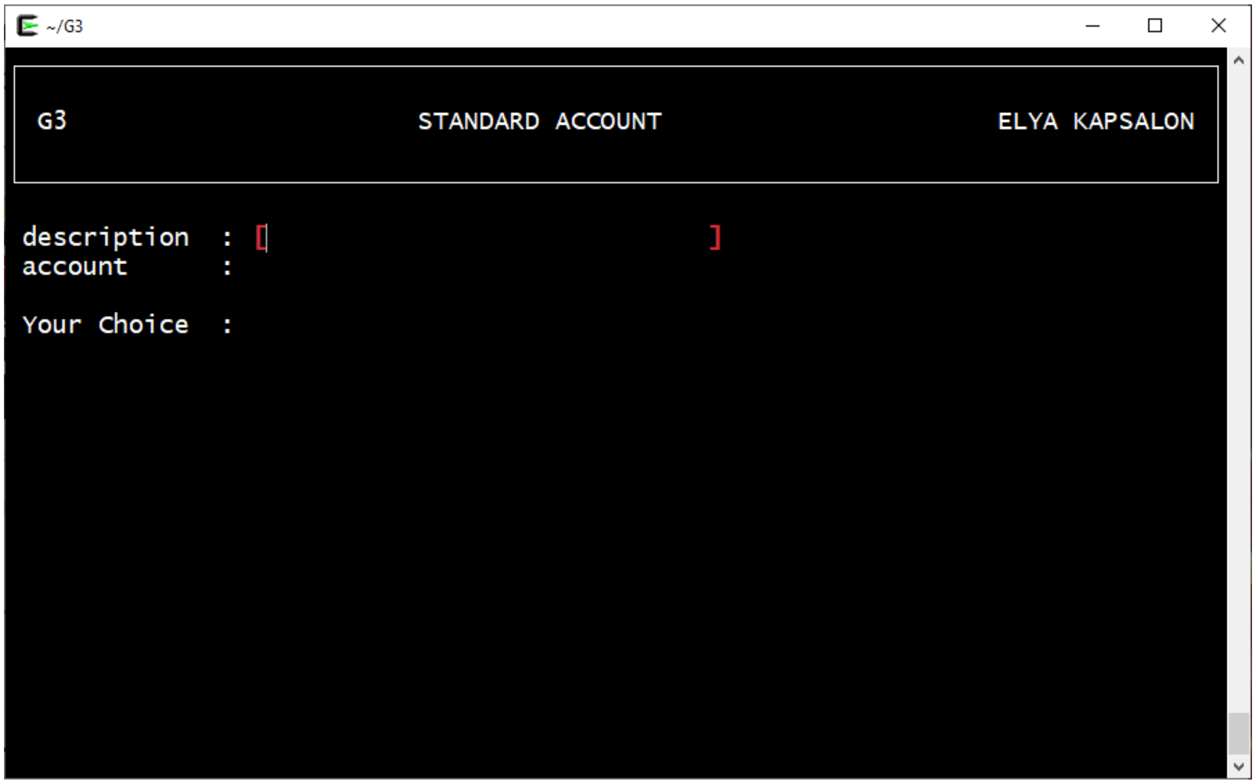
K



```
~/G3

 G3                       CREATE ACCOUNT                    ELYA KAPSALON

account      : [0                   ]
description  :
account type :
VAT code     :

Your Choice  :
```

Accounts can be created in this screen. It establishes a connection between an account number and a description. The account type must be “w” if it is a Profit & Loss account or “b” if it is a Balance Sheet account. The VAT code can be used to trigger the automatic split of a VAT amount. The code should be connected to both an account and a percentage.

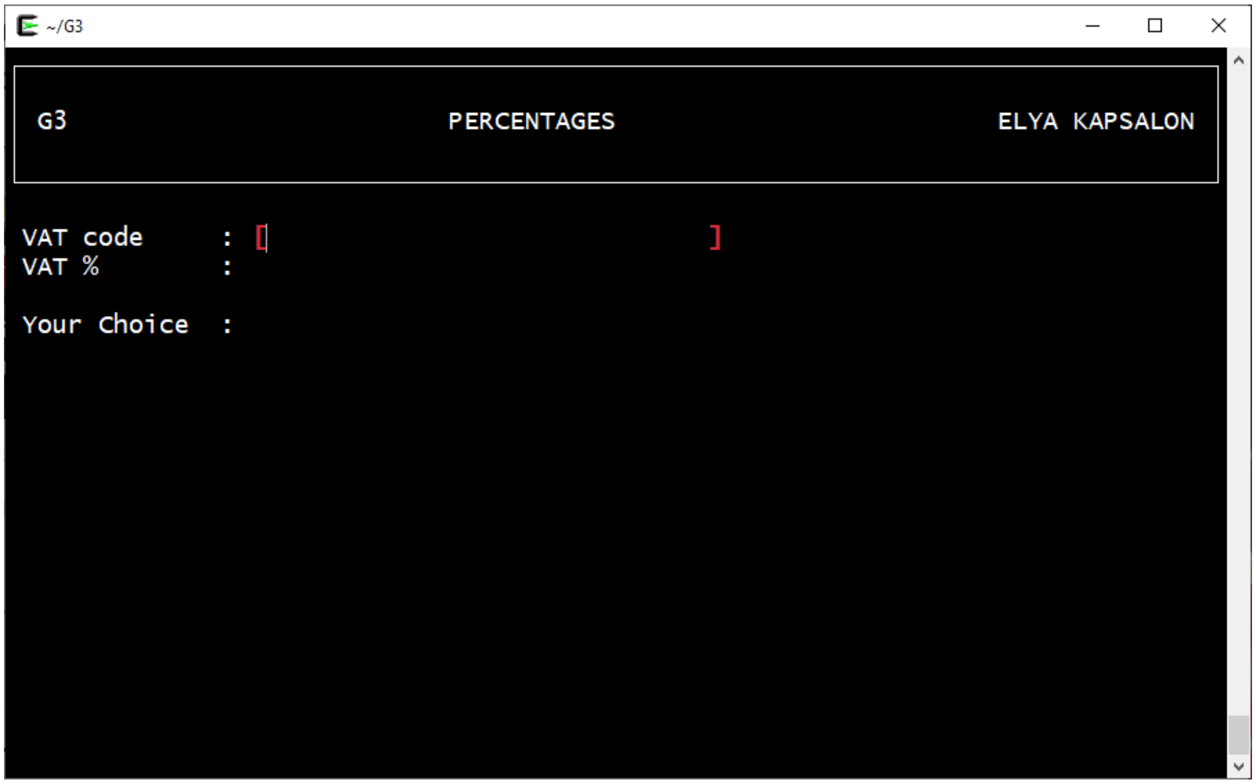
L



```
~/G3

 G3                    STANDARD ACCOUNT                ELYA KAPSALON

description  : [                              ]
account      :

Your Choice  :
```

The standard screen establishes a reverse index to the accounts. It serves two tasks: first it establishes the connection between Cash or Bank and an account and second it can be used to establish a connection between a VAT code and an account number. The description allows a maximum length of 29 characters. This limit comes from the standard 80 columns that a terminal screen allows to be displayed.

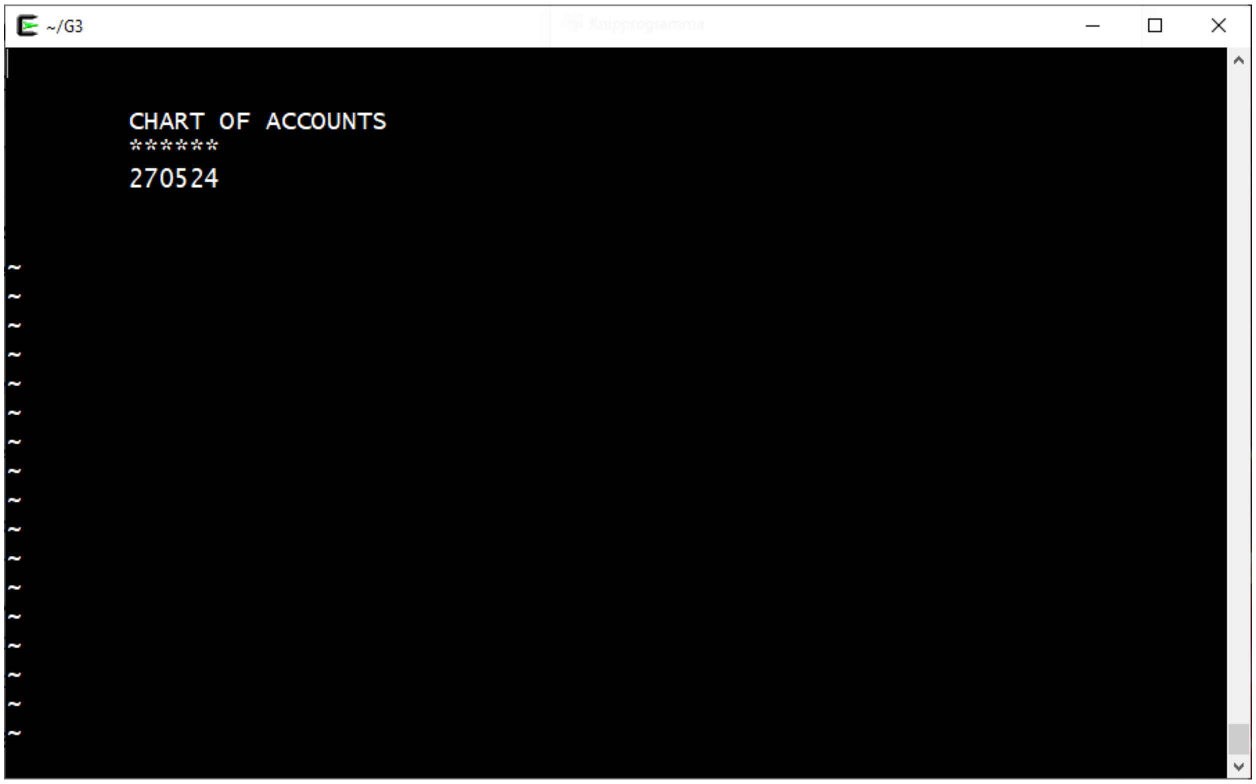
M



```
 G3                          PERCENTAGES                          ELYA KAPSALON

VAT code     : [                              ]
VAT %        :

Your Choice  :
```

This screen establishes the connection between a VAT code and a VAT percentage. The percentage should be given in two decimals.

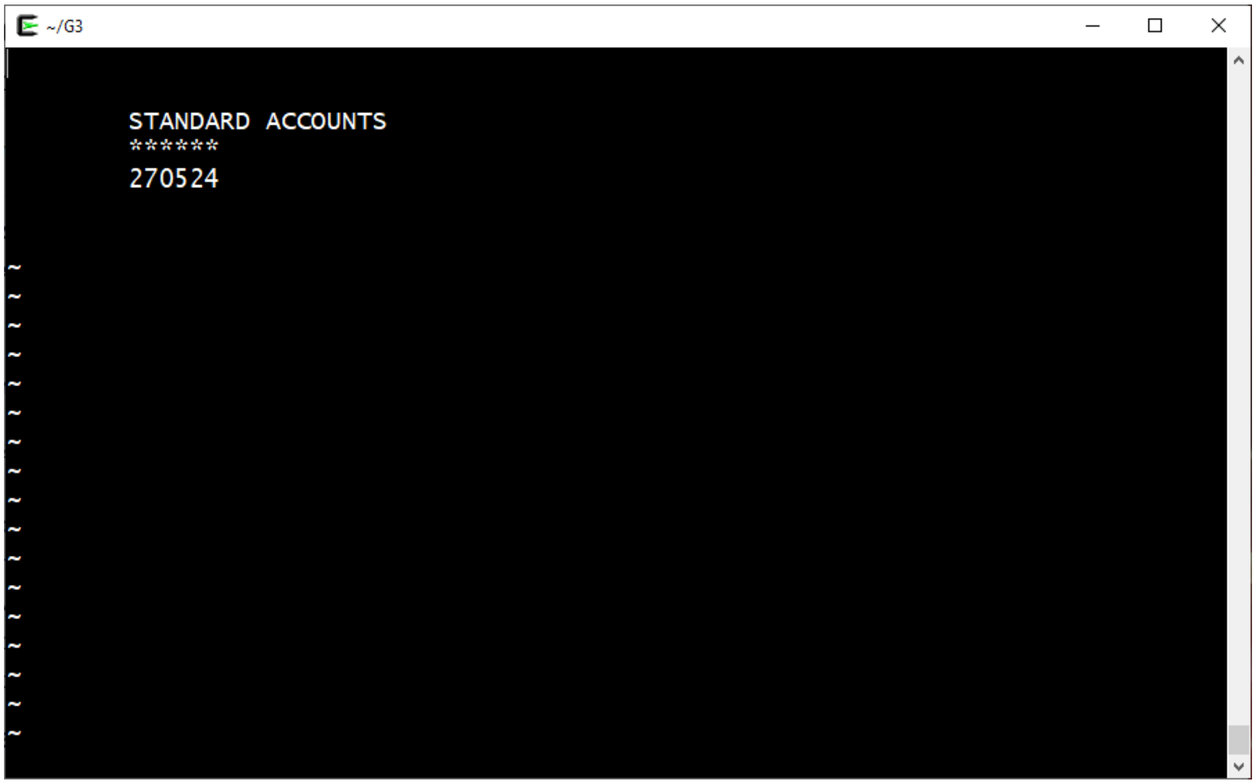
N



```
~/G3

        CHART OF ACCOUNTS
        ******
        270524

~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

This screen displays the Chart of Accounts. It is triggered directly from the menu, without an intervening selection screen. When no accounts have been created, the screen is empty, except for the header. The ~ characters show non-existing lines. They are inherited from some code that was used to implement the Kilo editor. The bottom line is left empty, reserved for messages.

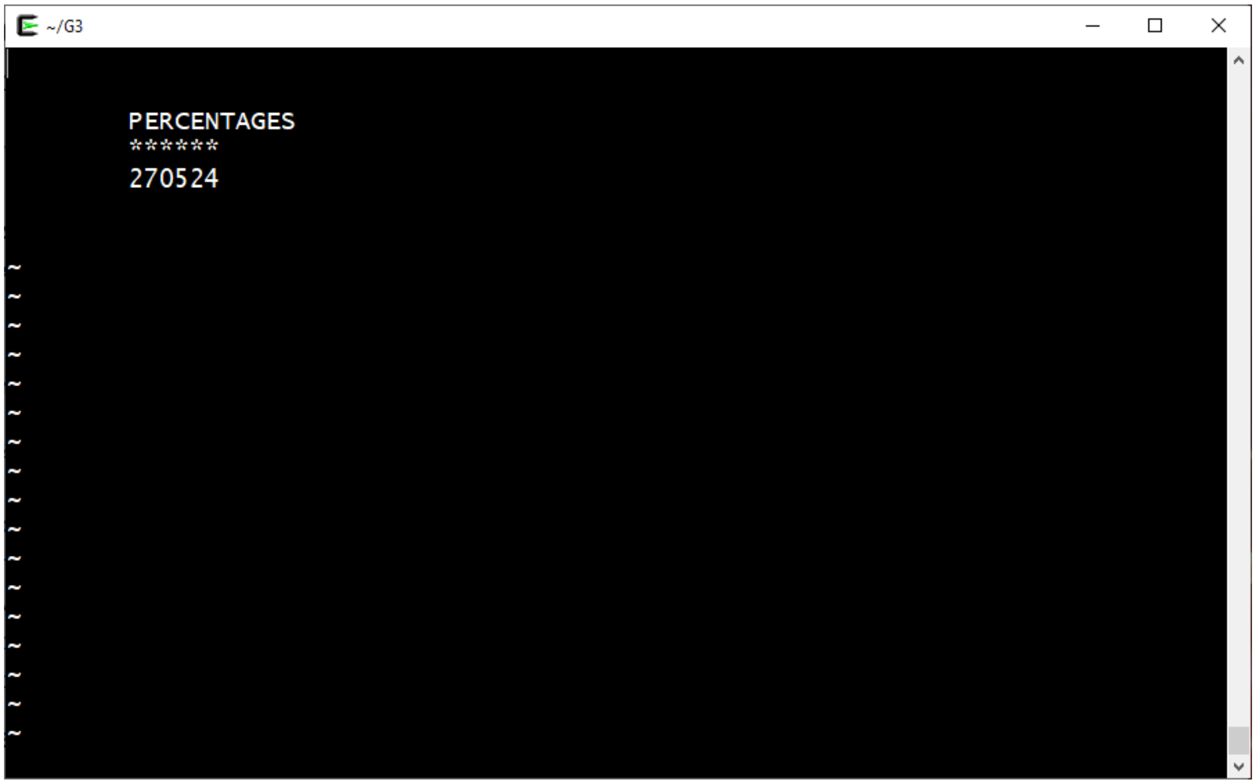
O



```
~/G3

        STANDARD ACCOUNTS
        ******
        270524

~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

This presents the standard accounts. Cash and Bank should be displayed here as well as all accounts that are connected to VAT codes.

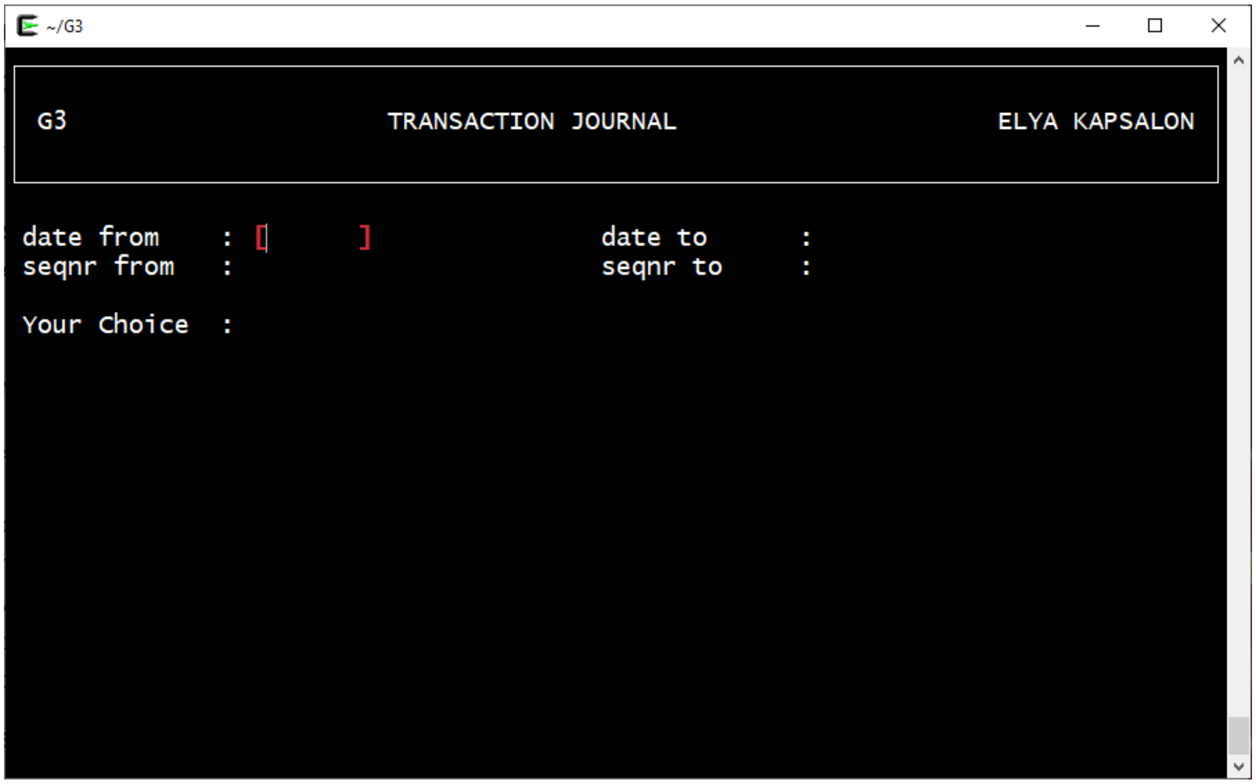
P



```
~/G3

        PERCENTAGES
        ******
        270524

~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

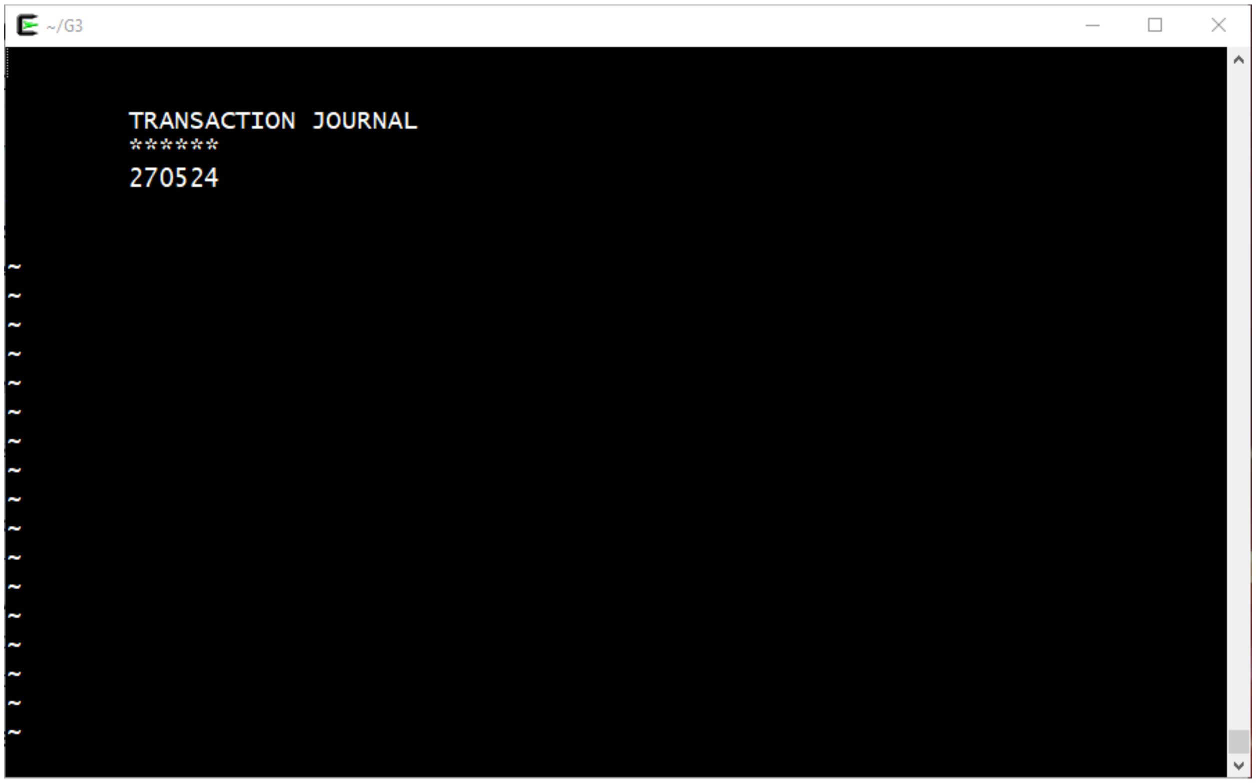
This screen should present the VAT percentages.

Q



```
~/G3

G3                     TRANSACTION JOURNAL                    ELYA KAPSALON

date from     : [       ]                 date to      :
seqnr from    :                           seqnr to     :

Your Choice   :
```

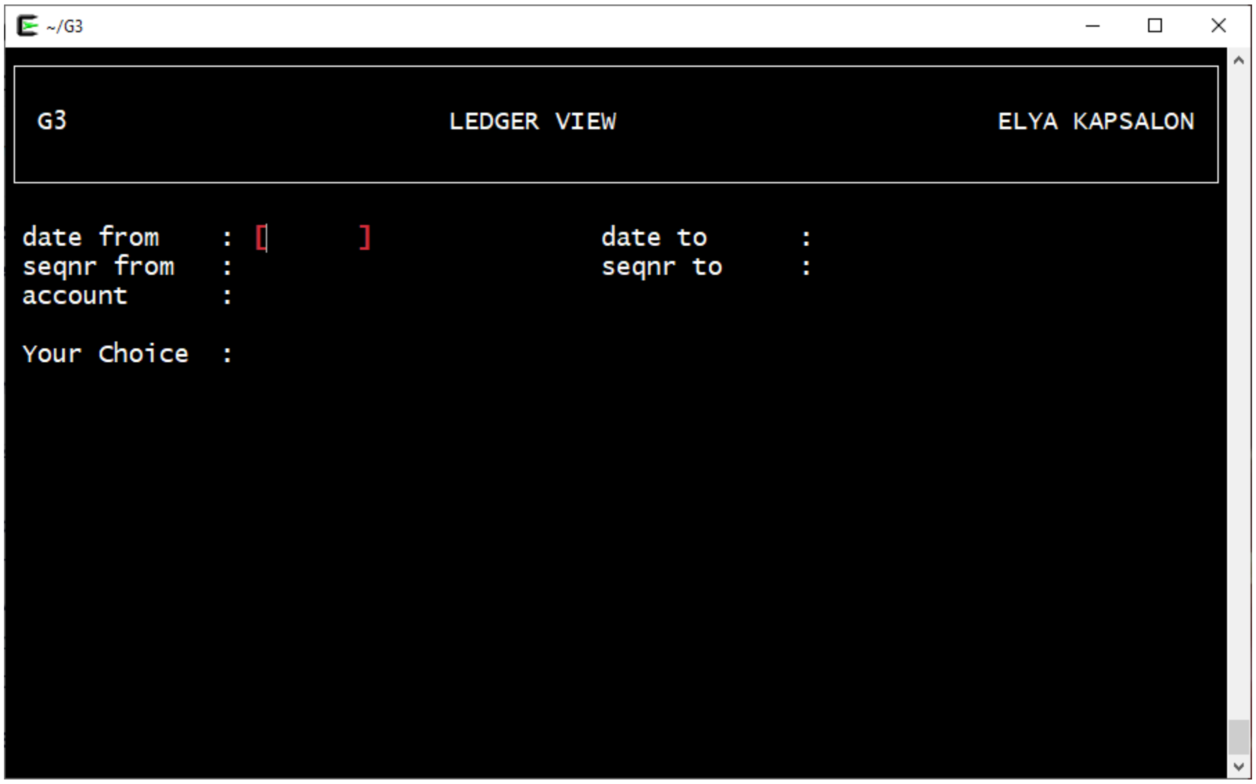
This is the selection screen before displaying the postings. The view can be delimited by date and/or sequence number.



```
        TRANSACTION JOURNAL
        ******
        270524


~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

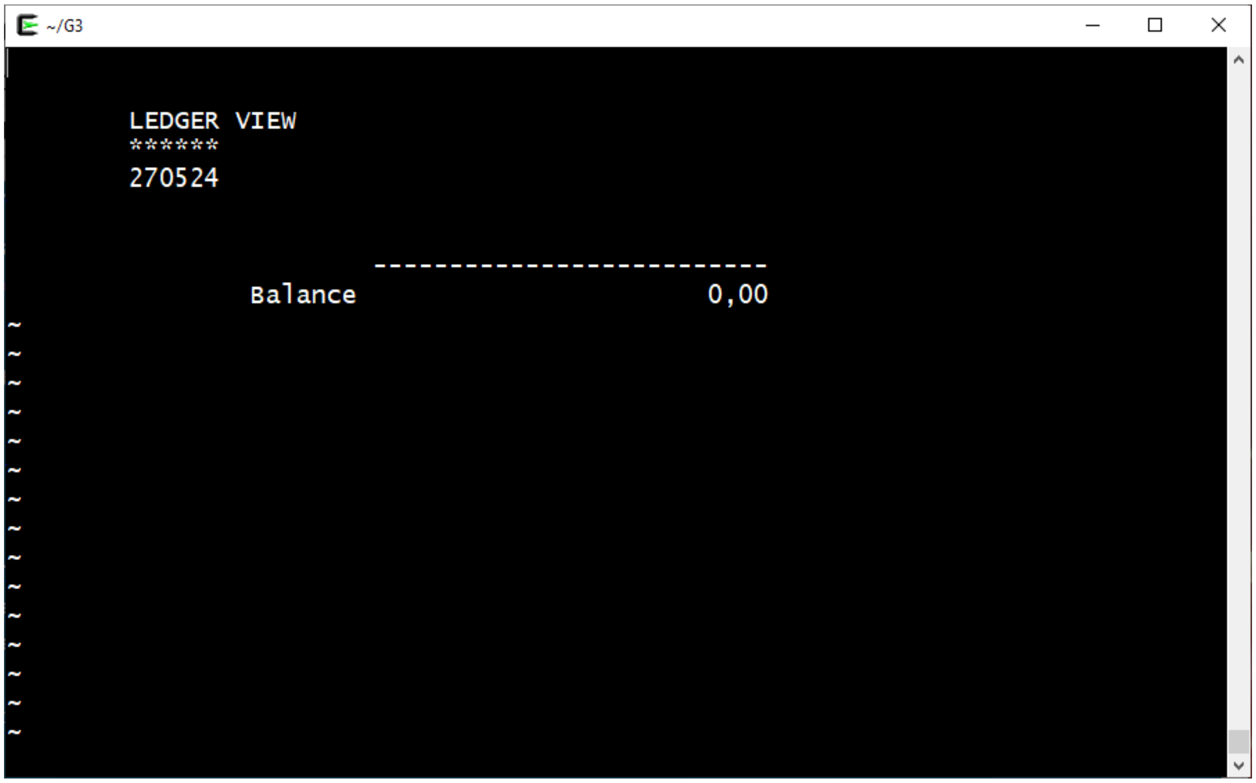
If no postings have been made, the screen will be empty.

R



```
~/G3

 G3                          LEDGER VIEW                     ELYA KAPSALON

date from    : [      ]               date to      :
seqnr from   :                        seqnr to     :
account      :

Your Choice  :
```

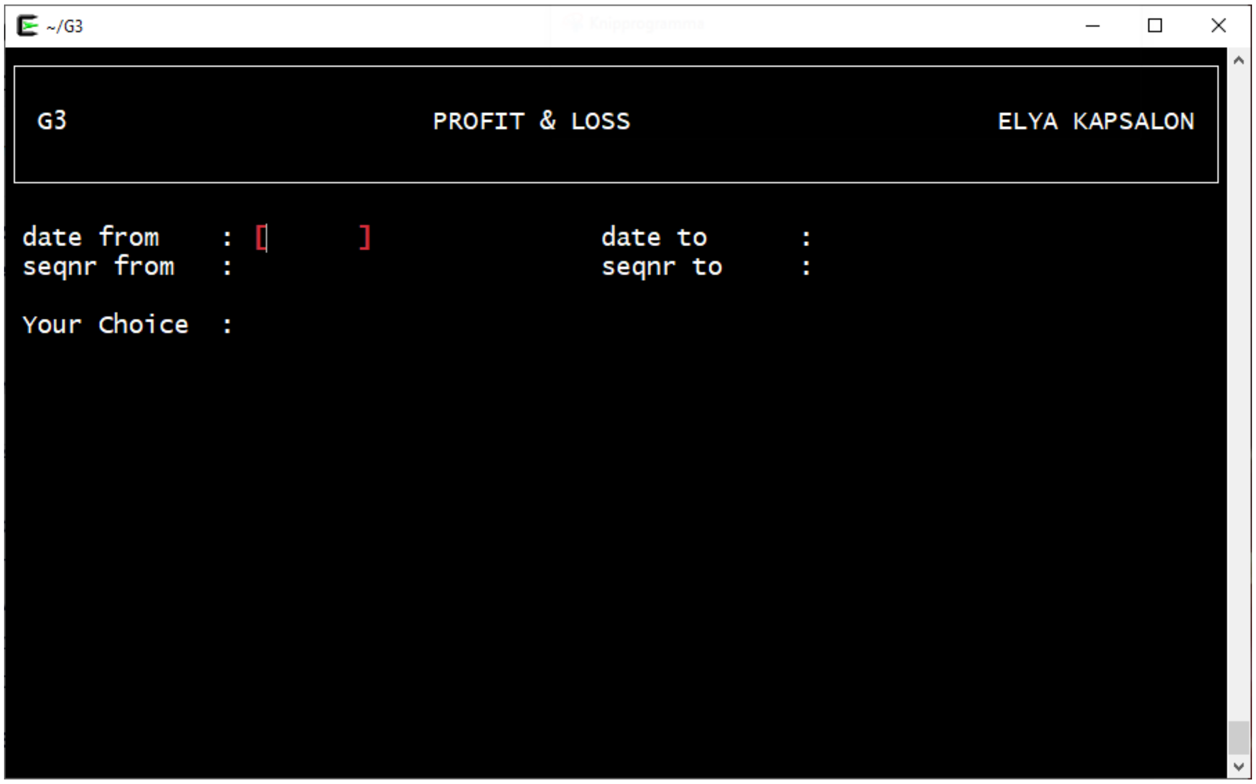
This is the selection screen of the Ledger View. The account number in this screen is mandatory. The view presents postings on one account only and also presents a summation.



```
~/G3

       LEDGER VIEW
       ******
       270524



              --------------------------
       Balance                      0,00
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

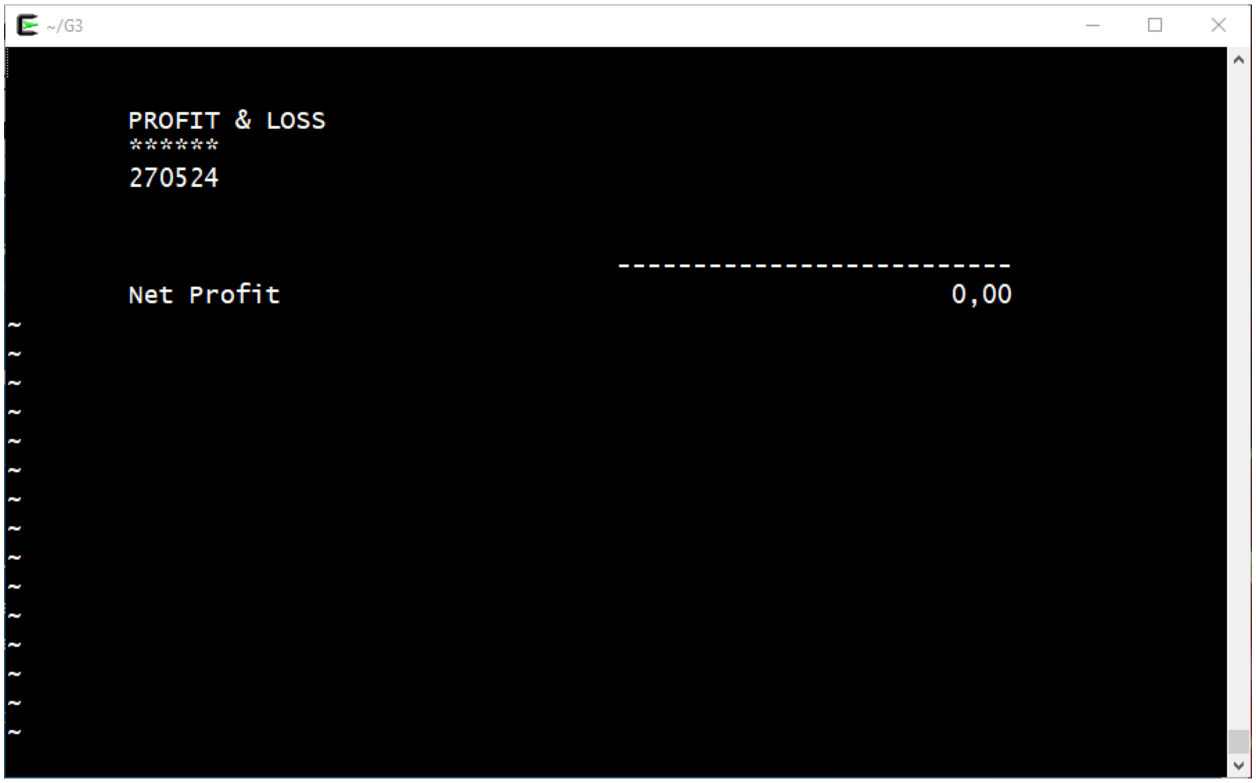
The Balance will be 0,00 when no postings have been made. Yes, amounts use a comma as decimal point. And the thousand separator is a full stop. Also, amounts can be very large. Computers are 64 bit now and that is why.

S



```
~/G3

 G3                        PROFIT & LOSS                         ELYA KAPSALON

date from    : [      ]             date to     :
seqnr from   :                      seqnr to    :

Your Choice  :
```

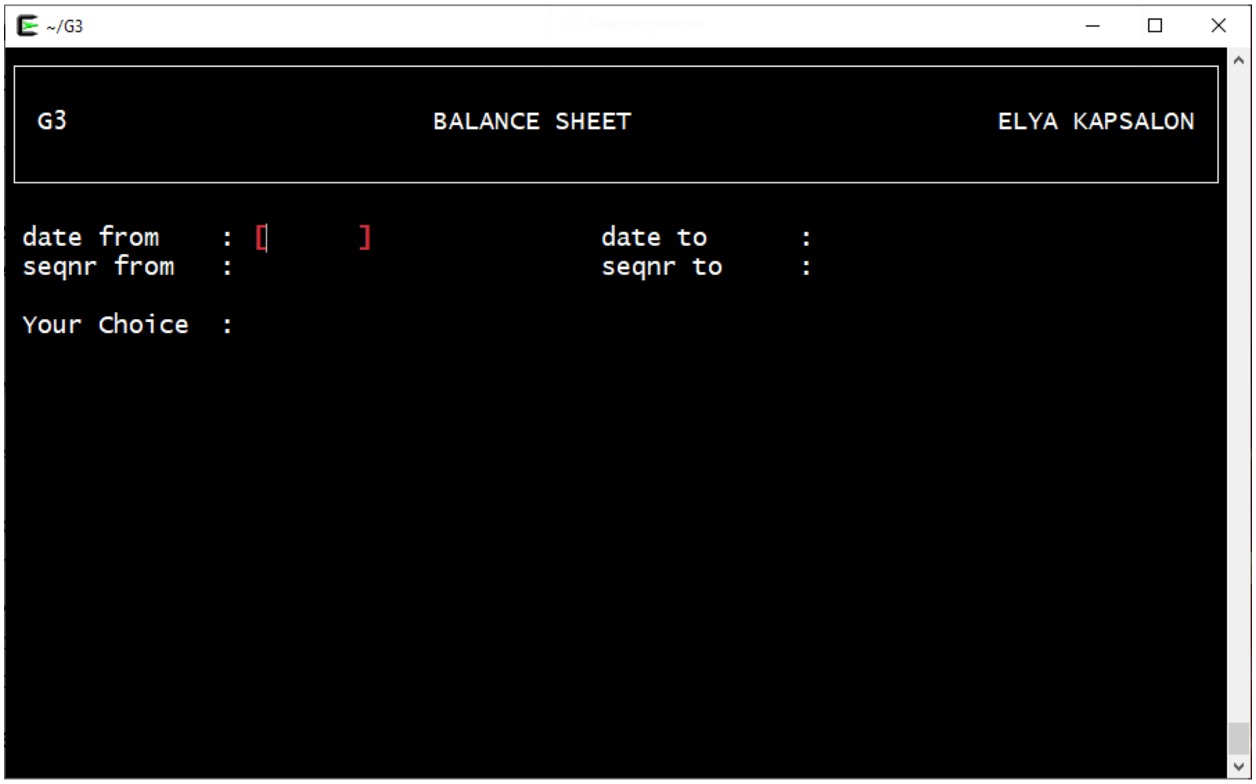
This is the selection screen of the Profit & Loss statement. For the current year it makes no difference whether the dates are filled or not. For previous years, after the Year End Closure of these years has been done, the result will be empty, unless the range of sequence numbers is filled in such a way that the Year End Posting is excluded. This will allow the inspection of the break down of the result amount into individual accounts.



```
        PROFIT & LOSS
        ******
        270524


                                                   --------------------------
        Net Profit                                                       0,00
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

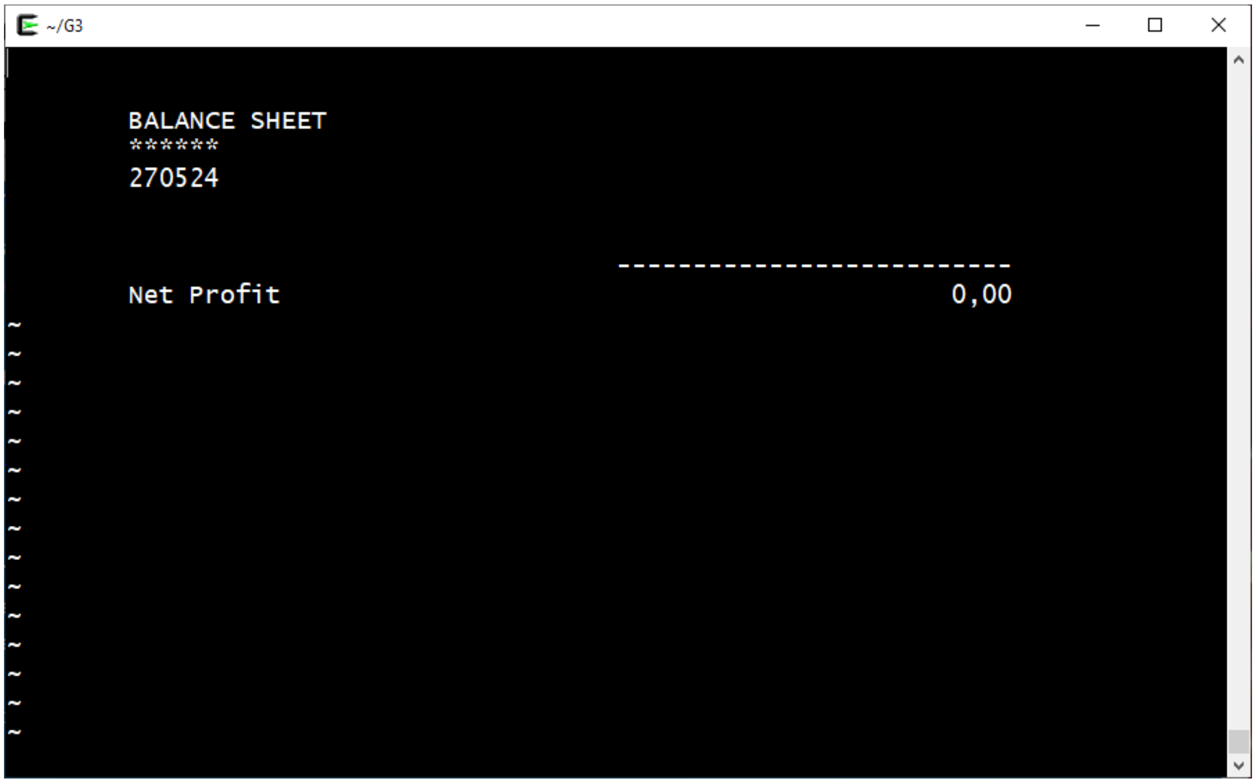
This is the Profit and Loss statement when no postings have been made. The amount presented may differ in sign from the Balance sheet account but should have the same text, in this case "Net Profit."

T



```
G3                        BALANCE SHEET                    ELYA KAPSALON

date from     : [      ]               date to      :
seqnr from    :                        seqnr to     :

Your Choice   :
```

This is the selection screen of the Balance Sheet. It is similar to the one of the Profit & Loss statement.

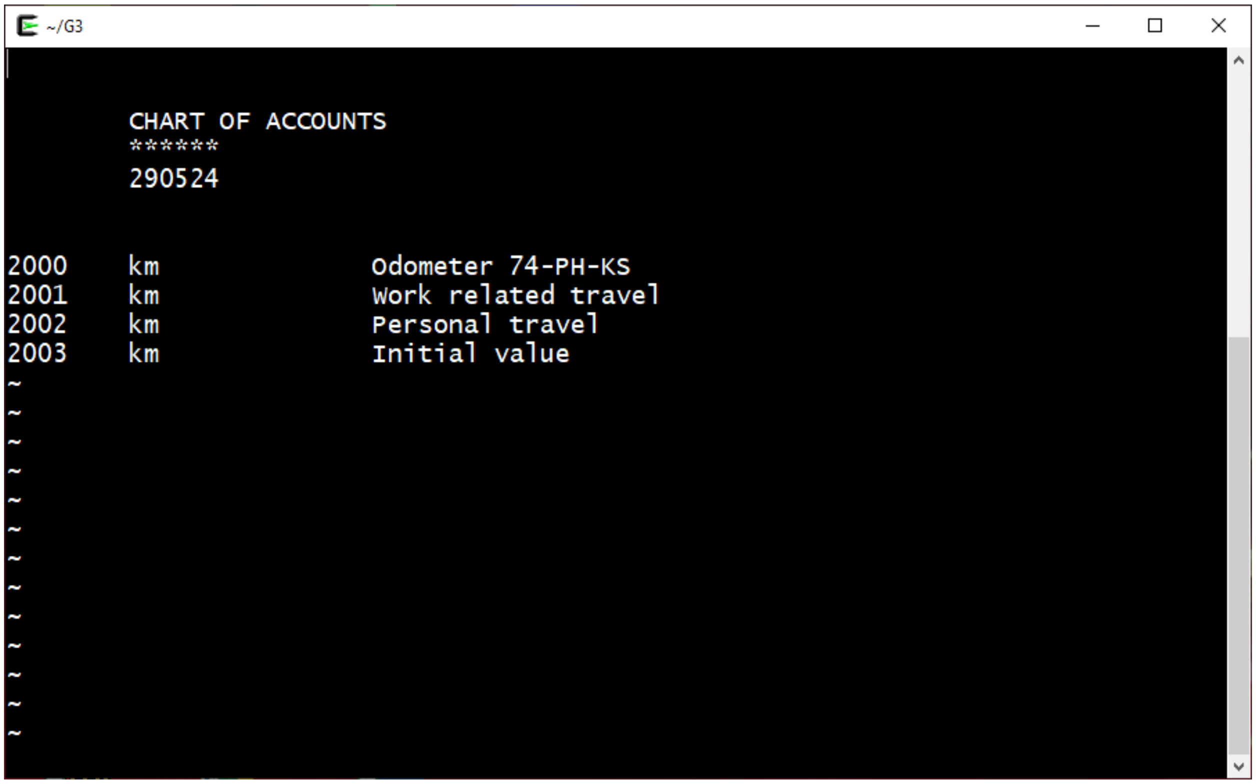


```
~/G3

        BALANCE SHEET
        ******
        270524


                                        --------------------------
        Net Profit                                            0,00
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

When no postings have been made yet, the result will be 0,00.

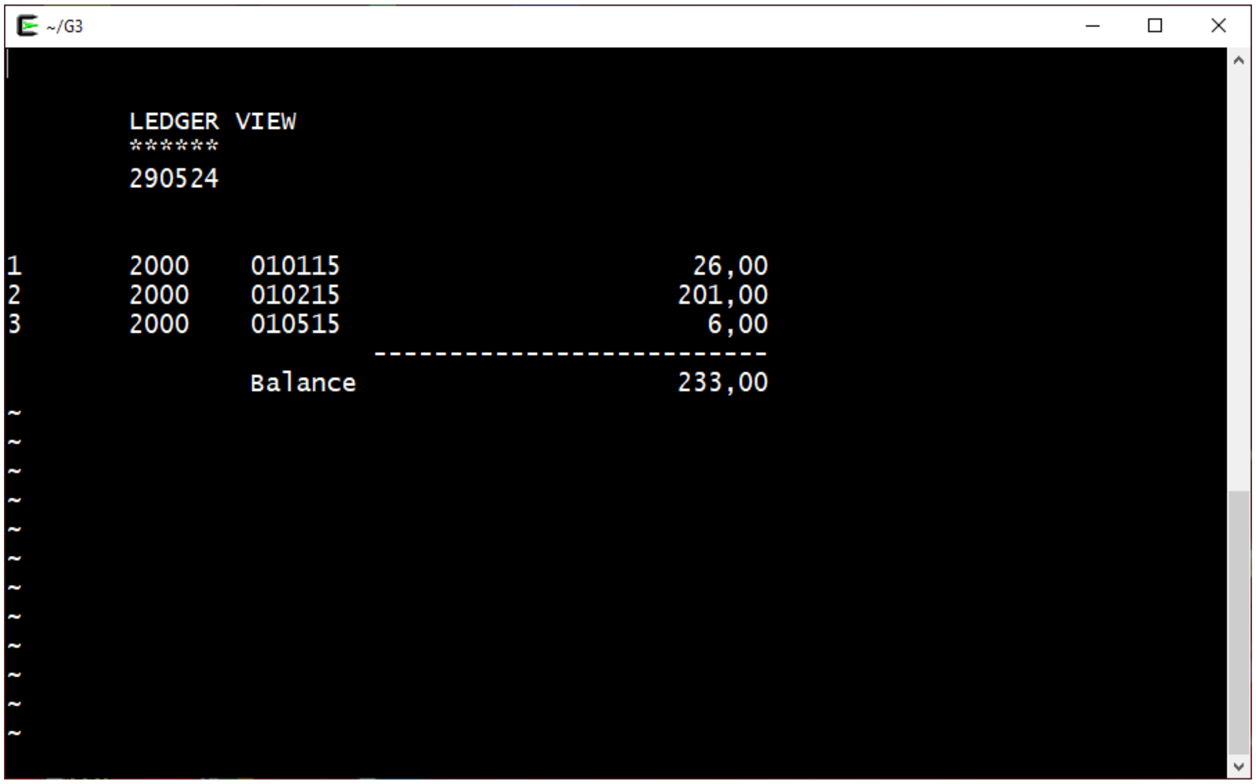
U

After all this preparation, both in the programming language and the application, it is time to create some postings. These first postings will not be financial postings. Instead, they register the odometer of a lease-car. Such registration is necessary if an employee wants to avoid that the value of the car is added to his income. He needs to sign a form and send that to the tax authorities, stating that the car will only be used for work related travels. In addition to that, the employee needs to register all travels made with the car. That means that for each travel the start value of the odometer needs to be written down, as well as the end value. From that, the amount of kilometers traveled can be calculated. The date must be recorded and the purpose of the travel. If an employee keeps a correct record, he is still allowed to travel 500 kilometers per year for personal use. This application can help with the registration. Copying the value of the odometer after engine start and before engine shut down will need pen and paper. That registration can then be used to enter in the computer as well as serve as proof of validity of the administration.



```
        CHART OF ACCOUNTS
        ******
        290524


2000    km              Odometer 74-PH-KS
2001    km              Work related travel
2002    km              Personal travel
2003    km              Initial value
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

Four accounts are needed, the odometer comes first, the next two are self-explanatory and the last one is needed because an odometer never starts at 0, not even if the car is new.

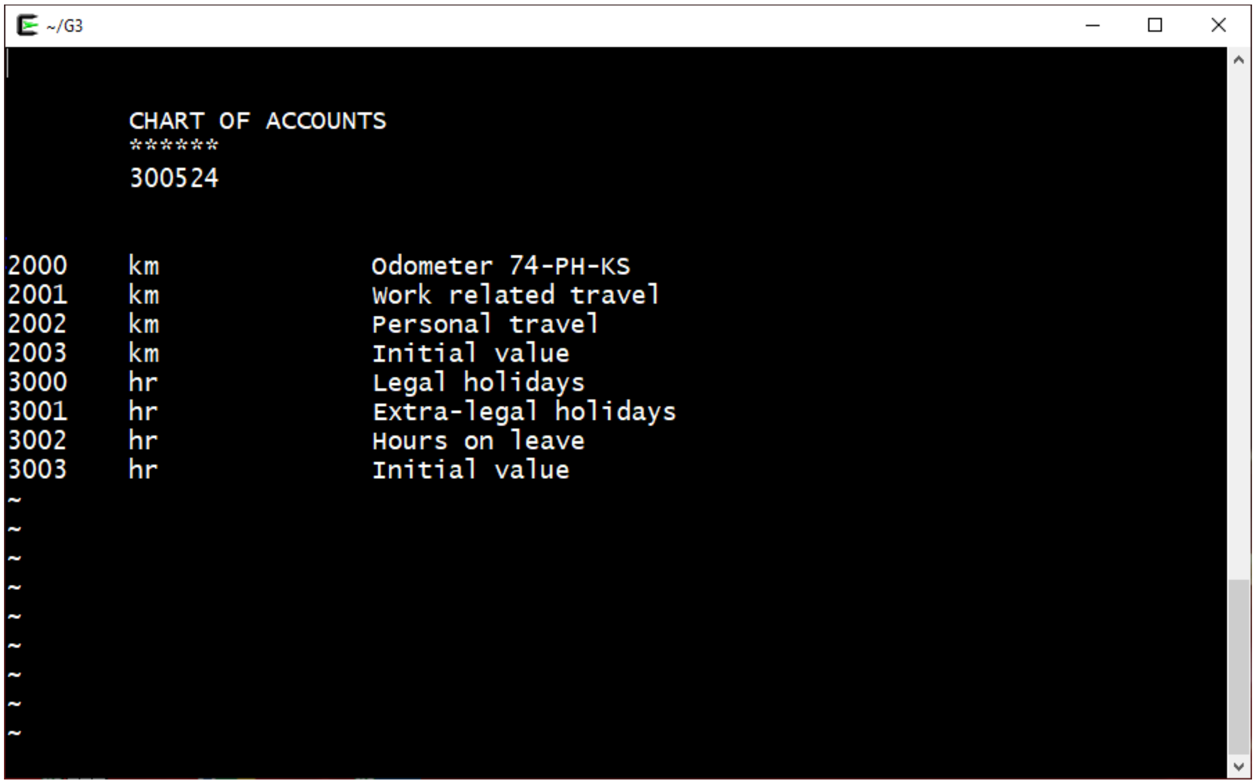


```
~/G3

        LEDGER VIEW
        ******
        290524


1       2000    010115                        26,00
2       2000    010215                       201,00
3       2000    010515                         6,00
                  -------------------------
                Balance                      233,00
~
~
~
~
~
~
~
~
~
~
~
~
```

These then are some postings. The initial value was 26 kilometers and registered in the test report that came with the car. The employee receives the car on February 1 and drives the car home. He then does not use the car and travels by train to his work. On May 1 there is a personal travel from home to the hospital and there it stops. The value of the odometer should then be 233. The decimals are always 00. The accounts 2001, 2002, 2003 will show negative values. The sign can be ignored.

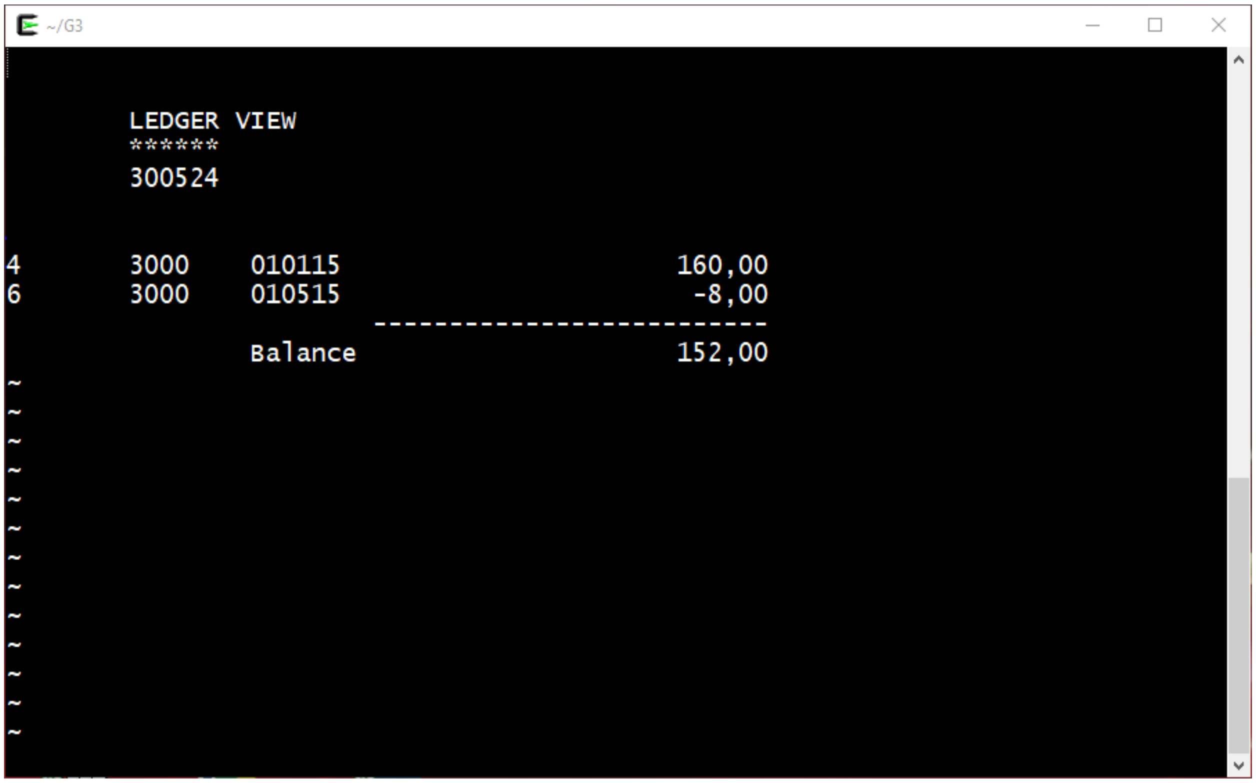
V

Another kind of non-financial accounting is the registration of holidays. Employees are granted legal holidays at the start of the year, and some extra legal holidays. The difference is that legal holidays have a shorter expiry date and need to be used first. Assuming 20 legal days and a contract of 40 hours per week this translates to 160 holiday hours. The Chart of Accounts does not come with a selection screen, so the account numbers of the previous chapter are shown as well.



```


        CHART OF ACCOUNTS
        ******
        300524


2000    km              Odometer 74-PH-KS
2001    km              Work related travel
2002    km              Personal travel
2003    km              Initial value
3000    hr              Legal holidays
3001    hr              Extra-legal holidays
3002    hr              Hours on leave
3003    hr              Initial value
~
~
~
~
~
~
~
~
~
```

The initial value is used to charge the legal and extra-legal holidays with a value, such that a count down is possible. Officially, these holidays are not granted at the start of the year, but for the purposes of recording holidays, it is convenient to do it this way. And if the employee leaves the company in the course of the year, some calculations are necessary anyway.

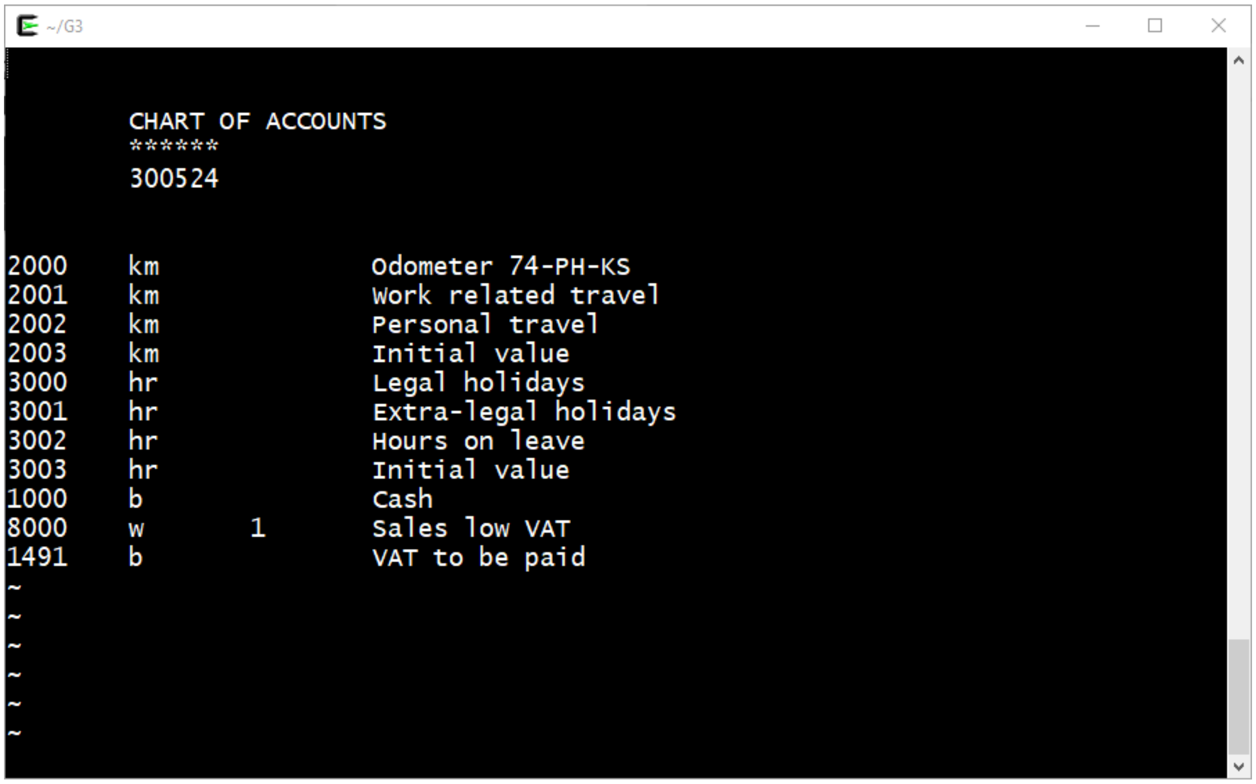


```
~/G3

        LEDGER VIEW
        ******
        300524


4       3000    010115                        160,00
6       3000    010515                         -8,00
                          --------------------------
                Balance                       152,00
~
~
~
~
~
~
~
~
~
~
~
~
~
```

And this is the view of the legal holiday hours after the employee has taken a day off on May 1. The remaining legal holidays are shown. When this reaches 0,00 any further holidays need to be taken from the extra-legal allowance.

W

It is now time for some financial accounting. A haircut costs 10 EUR and can be delivered at a reduced VAT tariff. It requires some setup before this posting can be made.

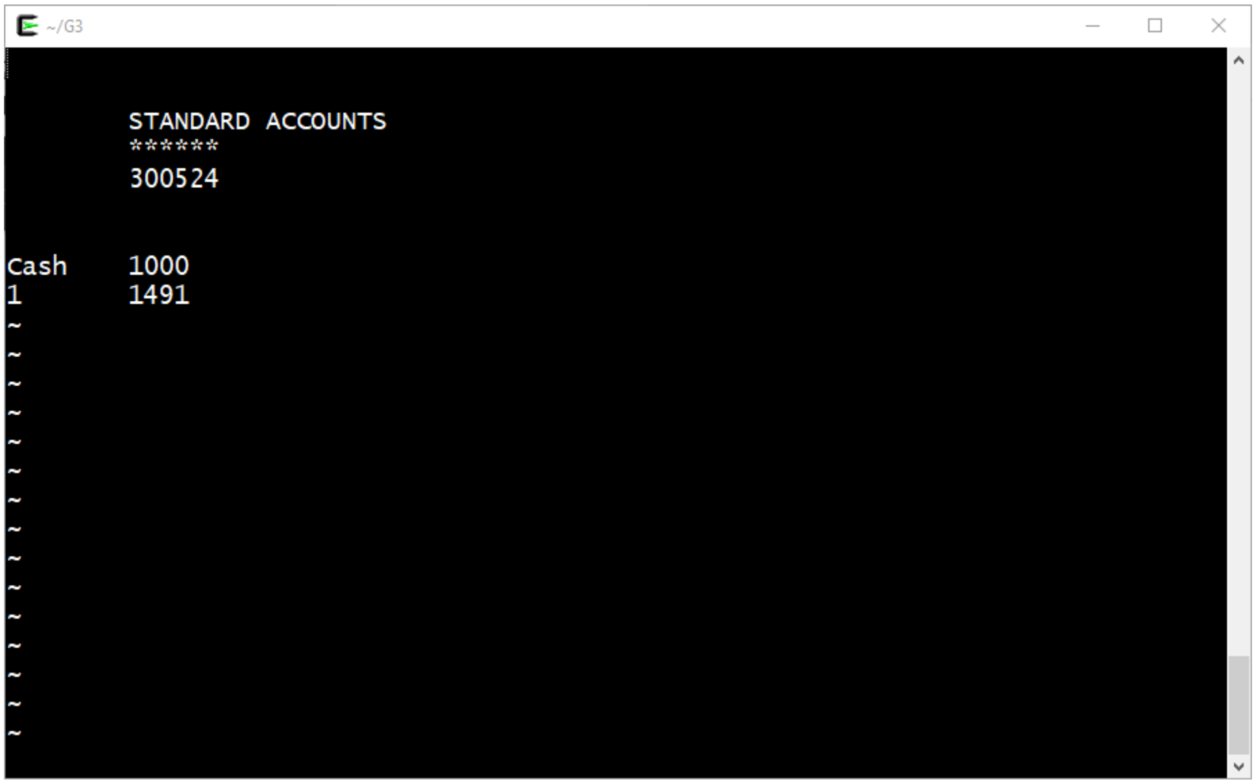


```
        CHART OF ACCOUNTS
        ******
        300524


2000    km              Odometer 74-PH-KS
2001    km              Work related travel
2002    km              Personal travel
2003    km              Initial value
3000    hr              Legal holidays
3001    hr              Extra-legal holidays
3002    hr              Hours on leave
3003    hr              Initial value
1000    b               Cash
8000    w       1       Sales low VAT
1491    b               VAT to be paid
~
~
~
~
~
~
```

The financial accounts must be characterized by either “b” or “w.” The “b” is for Balance Sheet accounts and the “w” is for Profit & Loss accounts. The third column shows the VAT code. The first digit of account numbers has some meaning to financial accounting:

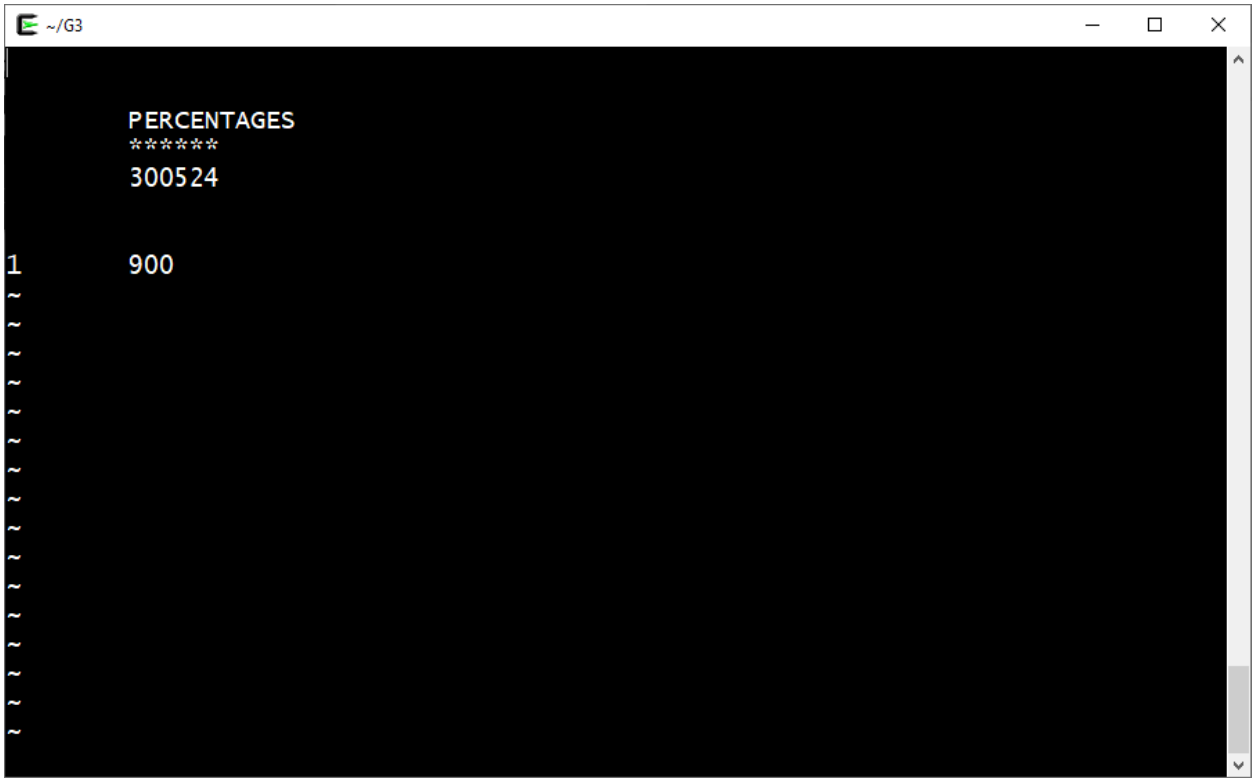
0 – assets
1 – liabilities
4 – costs
7 – stock
8 – revenue
9 – results

The VAT code needs to be connected to a VAT account and a VAT percentage:


~/G3

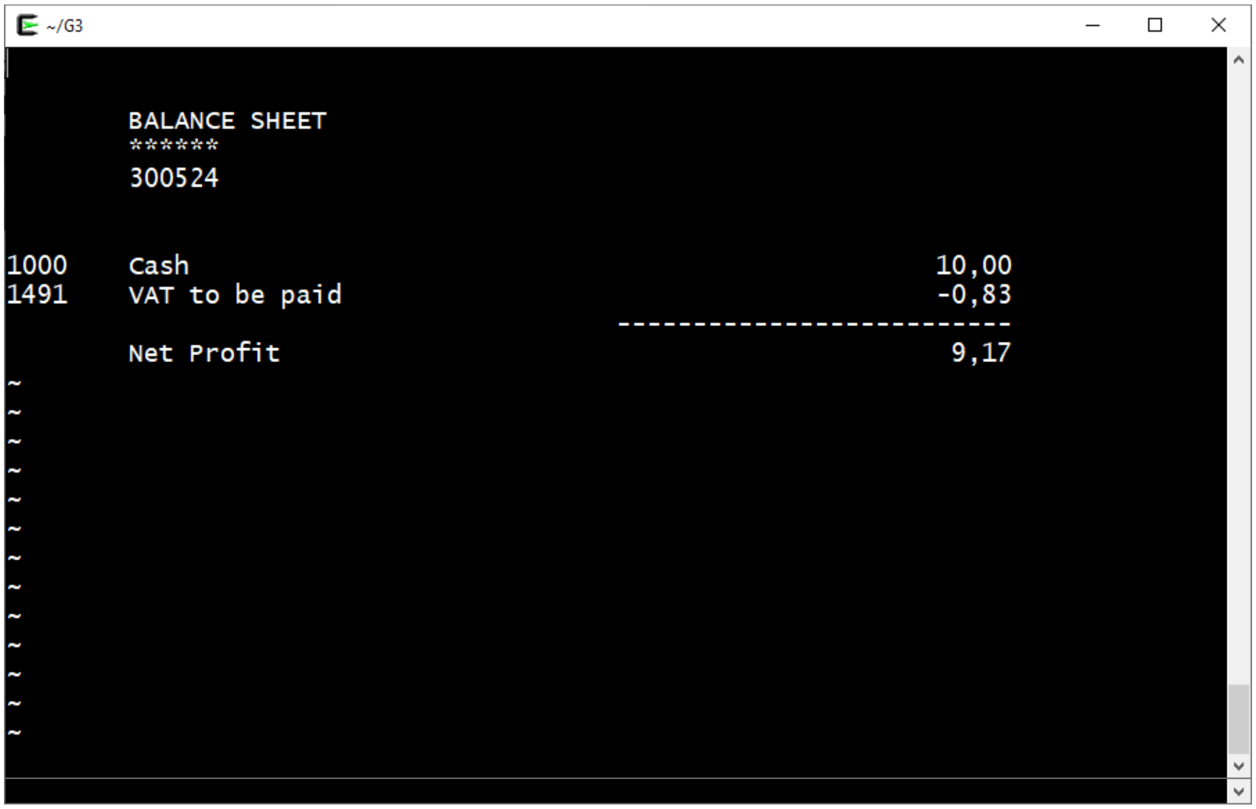
```
        STANDARD ACCOUNTS
        ******
        300524


Cash    1000
1       1491
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

The standard accounts also connect an account number to the Cash screen, such that in the Cash screen only one account needs to be specified. The reduced VAT percentage is 9.



```
        PERCENTAGES
        ******
        300524


1       900
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

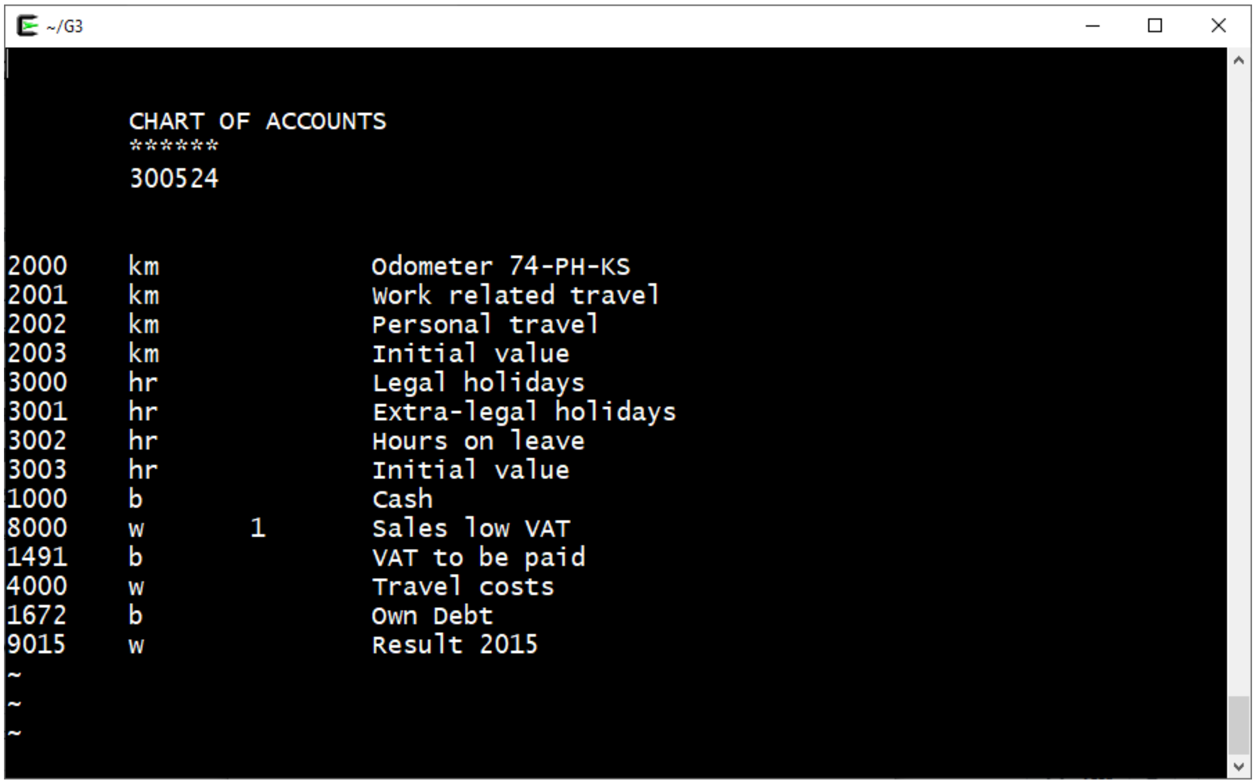
This shows the reduced VAT tariff, stored under the key 1.



```
~/G3

        BALANCE SHEET
        ******
        300524


1000    Cash                                                10,00
1491    VAT to be paid                                      -0,83
                                                  --------------------------
        Net Profit                                           9,17
~
~
~
~
~
~
~
~
~
~
~
~
~
```

The Profit & Loss statement can now be shown, as well as the Balance Sheet. They must agree.

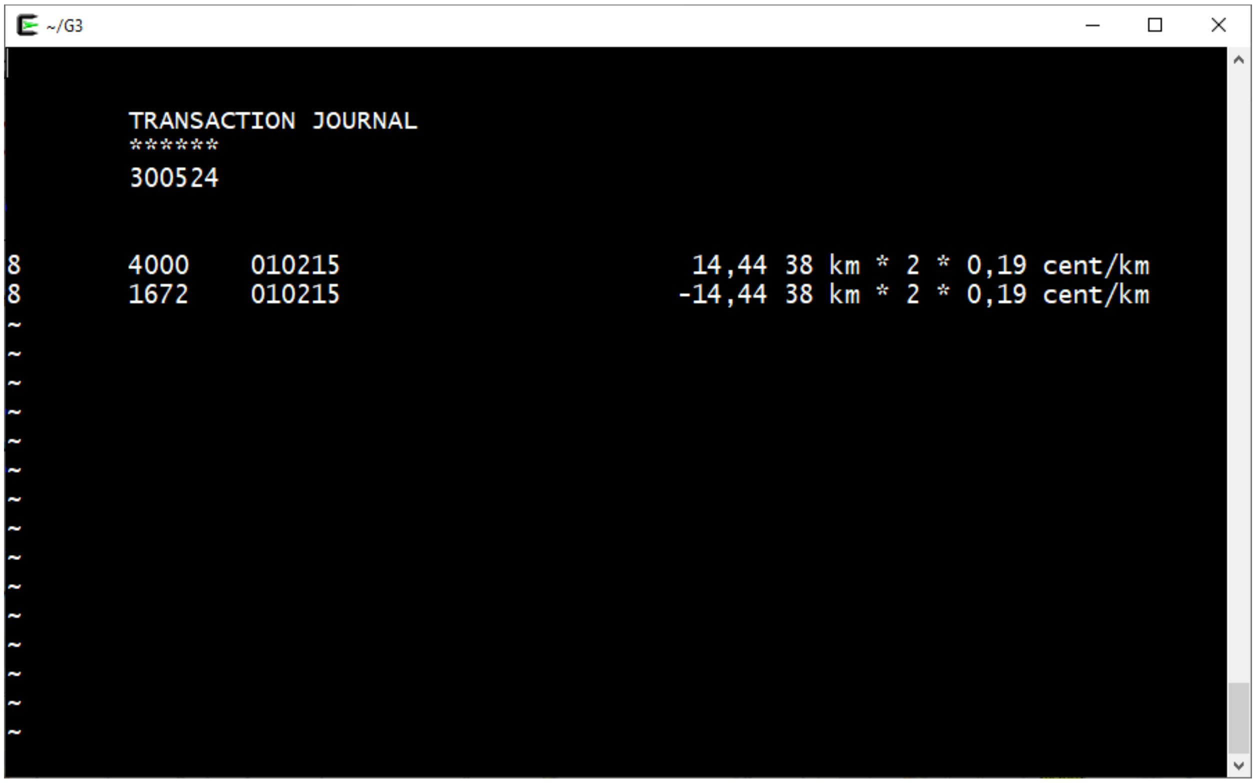
X



~/G3

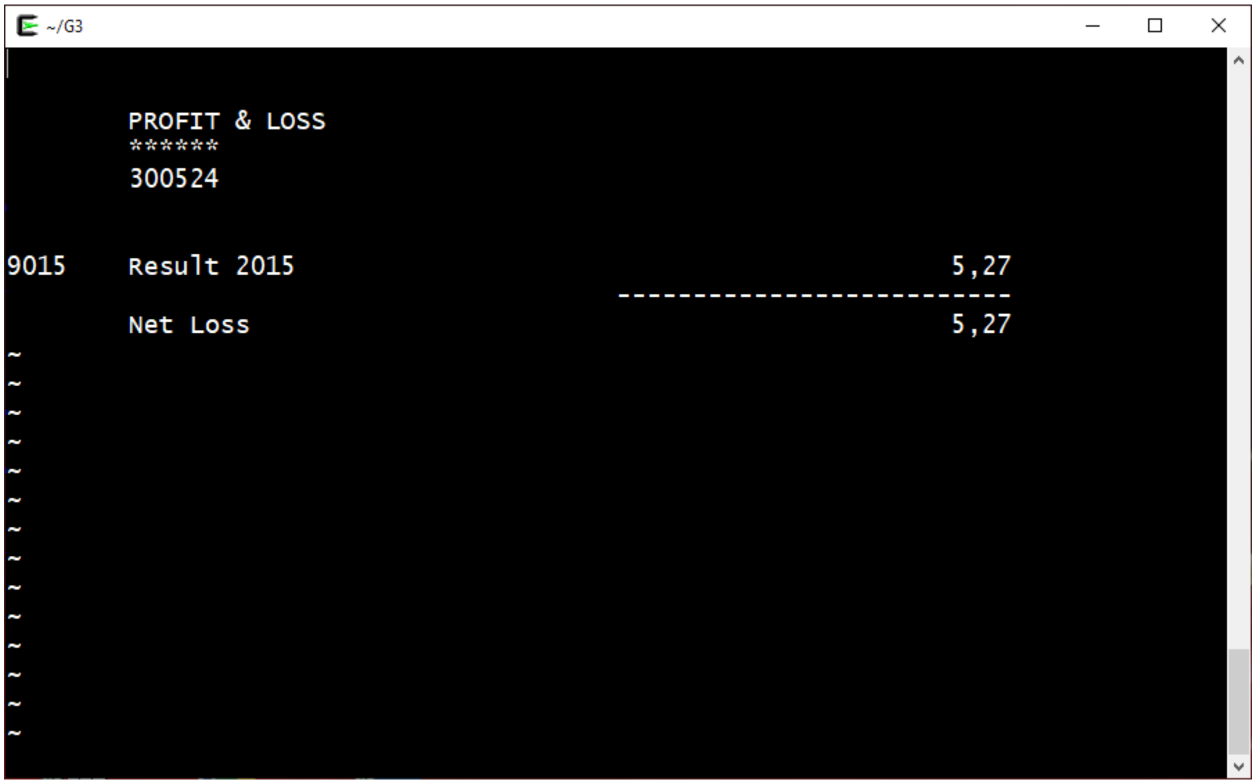
```


        CHART OF ACCOUNTS
        ******
        300524


2000    km              Odometer 74-PH-KS
2001    km              Work related travel
2002    km              Personal travel
2003    km              Initial value
3000    hr              Legal holidays
3001    hr              Extra-legal holidays
3002    hr              Hours on leave
3003    hr              Initial value
1000    b               Cash
8000    w       1       Sales low VAT
1491    b               VAT to be paid
4000    w               Travel costs
1672    b               Own Debt
9015    w               Result 2015
~
~
~
```

Before doing the Year End posting, some new accounts are needed.

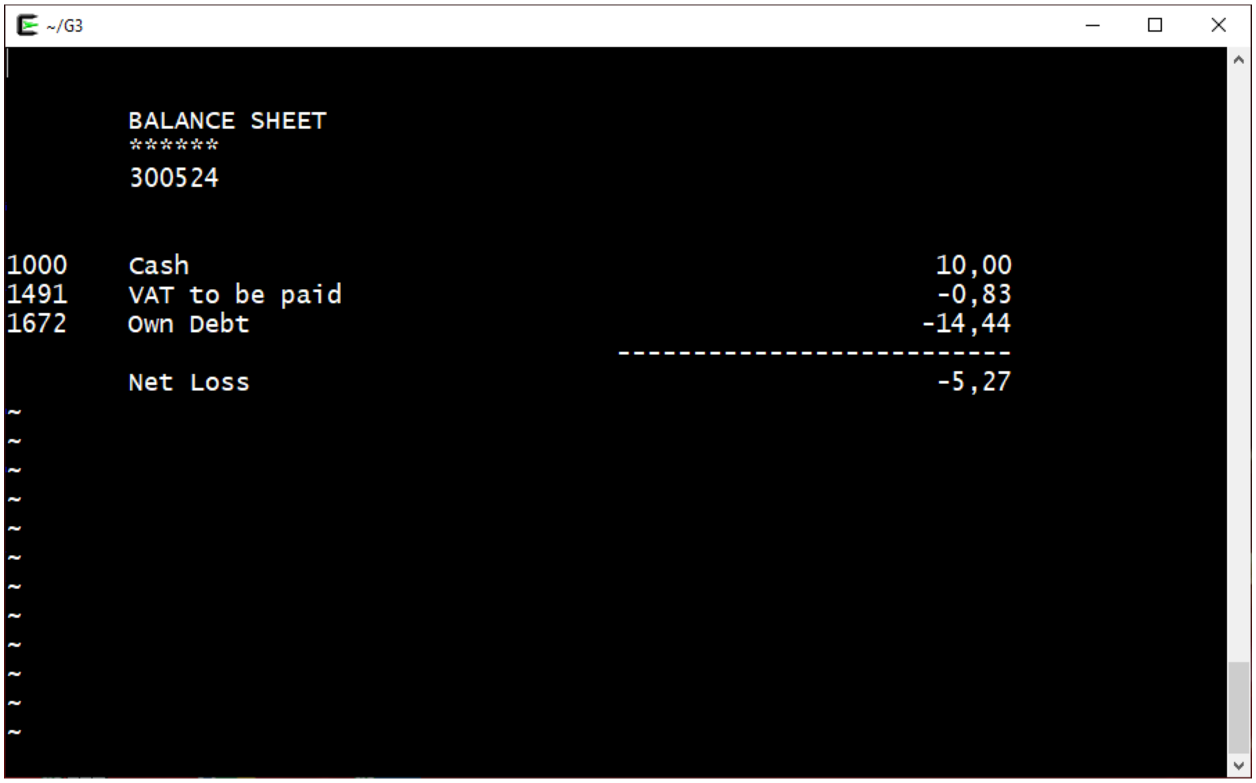


```
~/G3

        TRANSACTION JOURNAL
        ******
        300524


8       4000    010215                          14,44 38 km * 2 * 0,19 cent/km
8       1672    010215                         -14,44 38 km * 2 * 0,19 cent/km
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

Travel costs are rewarded with 19 cents per kilometer travelled, resulting in a loss.

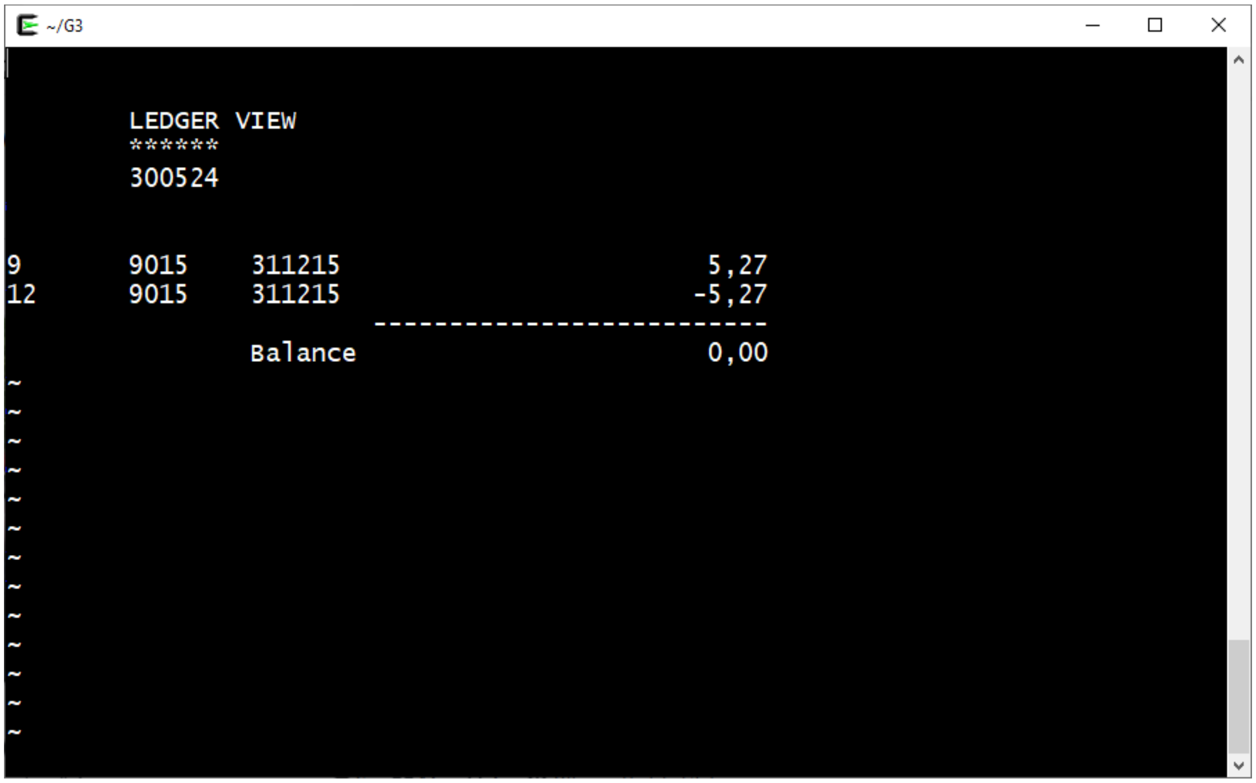


```
~/G3

        PROFIT & LOSS
        ******
        300524


9015    Result 2015                                       5,27
                                          --------------------------
        Net Loss                                          5,27
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```



```
~/G3

       BALANCE SHEET
       ******
       300524


1000   Cash                                              10,00
1491   VAT to be paid                                    -0,83
1672   Own Debt                                         -14,44
                                         ---------------------

       Net Loss                                          -5,27
~
~
~
~
~
~
~
~
~
~
~
~
```

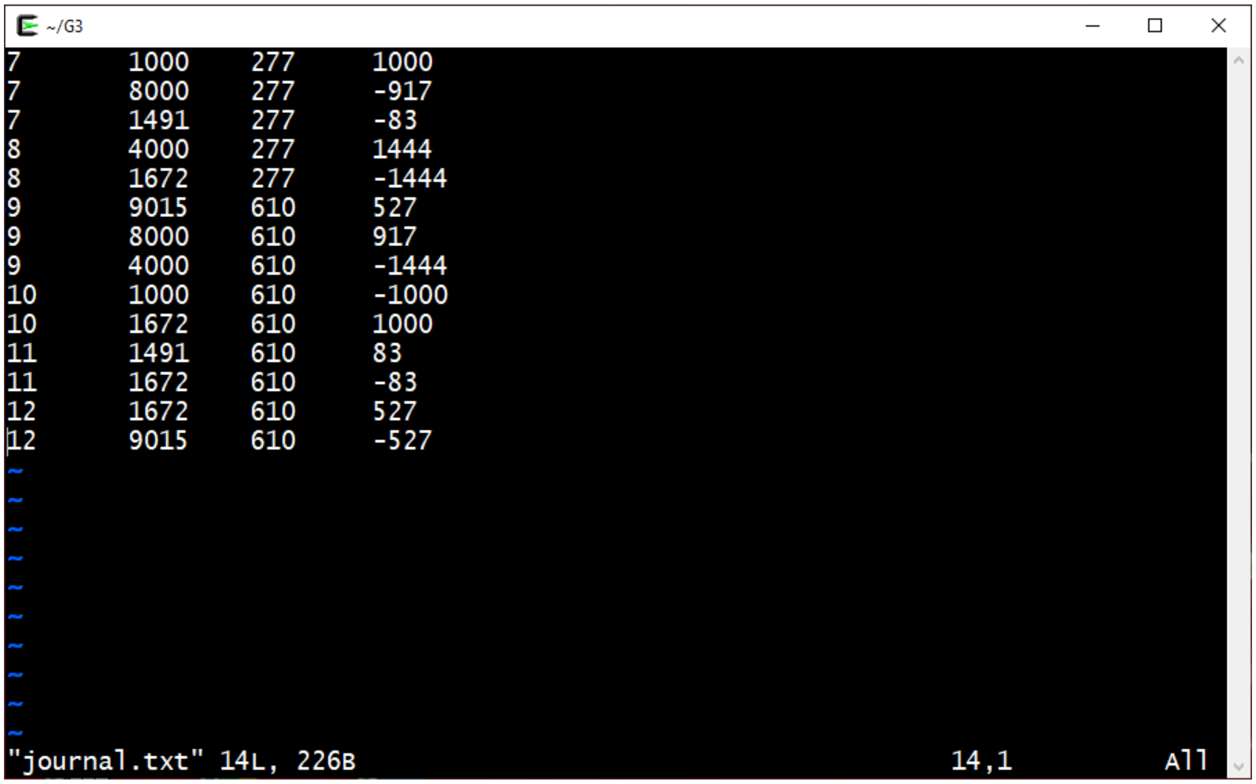
The Balance sheet agrees. The posting that filled the Results account was done by the menu item Year End Closing. This takes care of Profit & Loss accounts only. These Balance Sheet accounts also need to be levelled with the Own Debt account and last but not least, the Result account needs to be levelled with Own Debt. As a result of this procedure, all accounts will be zero again, ready for a fresh start in the new year.



```
~/G3

        LEDGER VIEW
        ******
        300524


9       9015    311215                            5,27
12      9015    311215                           -5,27
                  --------------------------
                Balance                           0,00
~
~
~
~
~
~
~
~
~
~
~
~
~
```

This summarizes the year 2015. Posting number 9 shows a loss of 5 EUR 27 cents.

Y

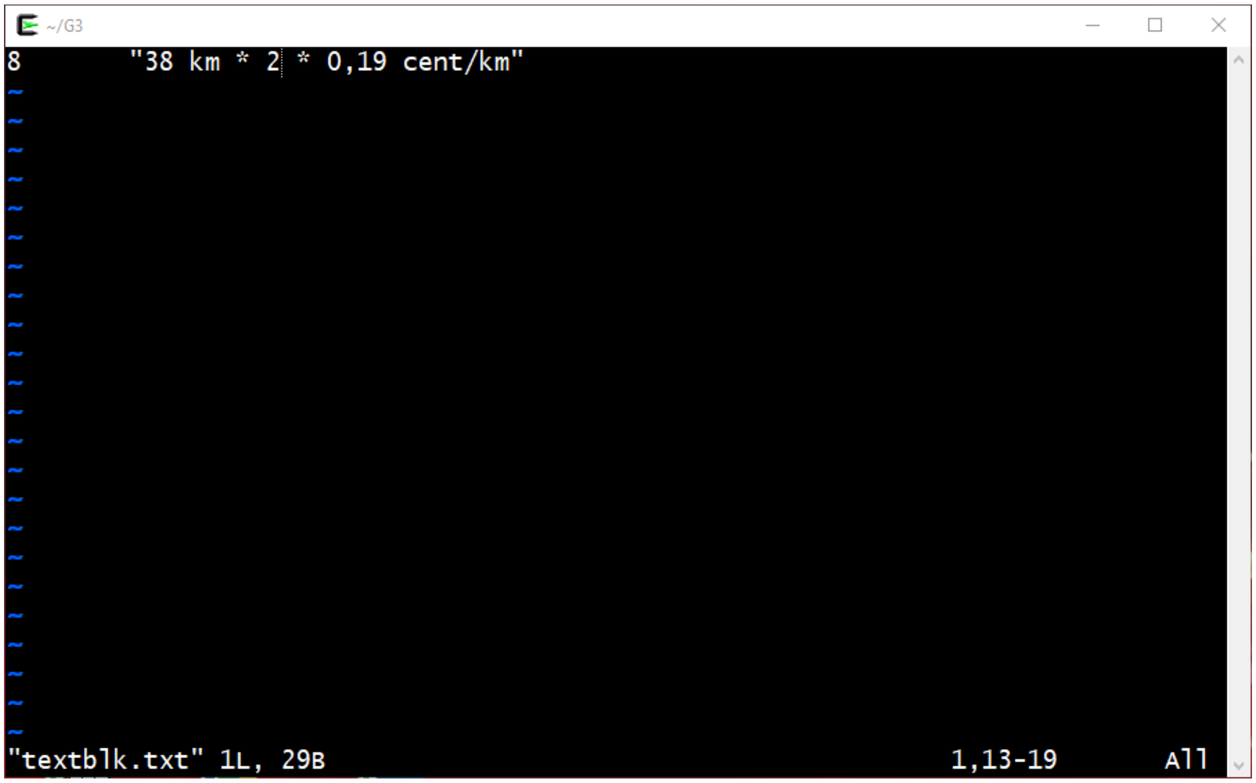


```
7        1000    277     1000
7        8000    277     -917
7        1491    277     -83
8        4000    277     1444
8        1672    277     -1444
9        9015    610     527
9        8000    610     917
9        4000    610     -1444
10       1000    610     -1000
10       1672    610     1000
11       1491    610     83
11       1672    610     -83
12       1672    610     527
12       9015    610     -527
~
~
~
~
~
~
~
~
~
~
"journal.txt" 14L, 226B                                       14,1          All
```

The data that is entered in the screens is stored in simple text files. Displayed is the file with transactions. In this view only the financial transactions are shown. This file is a true datafile in the sense that each line has a unique key, consisting of sequence number and account number. The files are:

account.txt
journal.txt
percent.txt
stdacct.txt
textblk.txt

All files are at the same time export files, easily loadable in a text editor or spreadsheet for further processing. All files are also auditfiles, tracking input, without modifications or deletions. All files allow logical modifications by adding corrections, that overrule earlier additions. There are no physical modifications or deletions.



```
8       "38 km * 2 * 0,19 cent/km"
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"textblk.txt" 1L, 29B                                         1,13-19       All
```

This is the file with comments, as part of the journal.txt. It shows the two datatypes that are stored: integer and string. Both are Joy source code. Also Joy source code is the file “lang.joy” that is read at startup and contains the texts that are displayed in menu’s and screens.

Z

Joy is an experimental programming language, trying to remove variables, seeing how far this can be stretched. The answer to that question is known: when implementing the quadratic formula, the lack of named parameters becomes painful.

G3 is an application, written in Joy. The interface dates back to the eighties and might look old. It so happens that accounting uses text and numbers and does not need graphics. Also, a small programming language such as Joy benefits from being able to interface with the terminal.

1

Addition. The comments in the screen print mention signed numbers. Numbers are taken from ℤ or ℝ and those sets already have signed numbers, so why is it explicitly mentioned that numbers are signed? A C programmer may know what is going on: overflow. C does not offer protection against overflow for performance reasons. Joy does away with the concern for performance but also has no protection. Joy is built on top of C, does not hide that, inherits most of the deficiencies from C and probably adds some of its own.

C also uses a call stack and offers no protection against stack overflow and neither does Joy. Modern operating systems, when confronted with a stack overflow, terminate the process that caused it. This can also happen to a Joy program. If it happens, the programmer needs to adjust the algorithm, using iteration instead of recursion. Joy has many recursion operators, a hobby of its designer, but those can only be used on small datasets. Joy was designed for small programs. If more data needs to be processed, some adjustments to the program may be necessary.

Addition is actually a feature that makes the use of a computer worthwhile. The accounting program in chapters A-Z makes use of it. It is possible to do accounting for a small company on paper, as was done from the 15th century onwards until the arrival of cheap personal computers. The many additions that need to be carried out are far better left to the computer. Data entry is also somewhat simplified compared to recording amounts with pen and paper.

And there is more to say about addition. As Gödel pointed out, every system of rules that contains addition is either incomplete or inconsistent or both. The propositional calculus does not have addition and is both consistent and complete. A progamming language is way more complicated than arithmetic and can be expected to have bugs in either the specification or the implementation or both.

Acknowledging that, the specification of the C programming language comes with undefined behaviour, unspecified behaviour, and implementation defined behaviour. Joy is no different in that respect. About the signed integer overflow: this is undefined behaviour. When this happens, the program loses meaning. In Joy it could be detected by doing unsigned integer addition first, and add the sign later on. Unsigned addition is well-defined. Ok, so there will be overflow. Then what? Change operands to bignums and then do the addition, with loss of speed? Or convert the operands to floats and do the addition with loss of precision? Either way, the program may not be what the programmer or user expects it to be. Also, Joy does not have bignums. It only has the bignum type, no more.

2

Multiplication. The tutorial of 42minjoy has some nice examples:

```
2 2 + 2 2 * =.
```

This results in `true`. Does that mean that `+` and `*` are the same? Another example is:

```
0 0 + 0 0 * =.
```

But these are probably the only examples. The example in the tutorial is sufficient to show that `+ ` and `*` are not the same:

```
2 3 + 2 3 * =.
```

This results in `false`. In general if two programs are suspected to give the same result, they need to be tested on all possible input. And that is a bit of a problem, even on modern computers. It is easier to disprove sameness.

The other example in the tutorial is:

```
6 6 * 5 7 * >.
```

This shows that a square that has the same circumference as a rectangle embraces a larger area. It is known that a circle has the best area/circumference ratio and a square can boast closer proximity to a circle than a rectangle.

The reasoning about square and rectangle associates mathematics with objects in the real world and that is a little dangerous. Mathematics can operate perfectly without any reference to the real world. It is a formal system. Associating multiplication with areas is possible, but in the end, multiplication is just repeated addition.

The example of areas has some historical significance. It can explain the abhorrence of negative numbers. After all, there are no negative areas. The number 0 was introduced in Western Europe around the year 1000, and again around the year 1200, but when Luca Pacioli codified the existing Venetian accounting practices around the year 1500, there was still a strong resentment against negative numbers. The accounting program in chapters A-Z uses them.

3

Equality. This is not at all an easy concept. Joy lumps together all datatypes that are considered numeric and allows them to be compared and lumps together all datatypes that have a string presentation and allows them to be compared. What remains are FILE pointers that can only be compared with FILE pointers and lists that cannot be compared with the simple operators.

But there is a problem, or rather a dilemma, because no matter how the problem is solved, it remains a problem. It is simply mitigated to somewhere else. Consider the comparison of a string and a symbol:

“plus” [plus] first =.

They compare equal. Apart from the double quotes they also look equal. But now consider this comparison:

“+” [+] first =.

Again, they look equal, but the comparison tells they are different. This solution was needed, in order to allow the grammar library to operate without problem. In that library \`+\` is not the addition operator, but a regular expression operator with the meaning one or more times.

“plus” [+] first =.

Now this comparison tells they are equal, even though they look different. At least they are pronounced the same. But what kind of sameness is used? Is it the looks or is it the pronunciation? Well, sometimes it is the one and sometimes the other. All symbols in the symbol table that do not start with an alphabetic, like \`+\`, have a nickname that is used in the comparison with strings. The nickname is sometimes the same as the name of the C function that implements the functionality of the symbol, but not always. This can lead to surprises.

“ast” [ast] first =.
“*” [*] first =.
“ast” [*] first =.

The same lines for multiplication, or Kleene star in the grammar library. The outcome of these three programs is: true, false, true.

4

Comparison. Comparison assumes that the datatype is ordered. Sets are only partially ordered. That means that both the following programs return false:

```
{1 2} {3 4} >.
{3 4} {1 2} >.
```

The comparison operators, in case of sets, are used for (strict) super/subset.

The `>` operator is the first user-defined function in 42minjoy. The only builtin that 42minjoy uses is `<`. In the original Pascal source `<` can be used for integers or strings. In the C source, the function strcmp must be used to compare strings. Anyway, here is the definition of `>`:

```
> == swap <
```

Indeed, if A < B, then it must be true that B > A. It can happen that parameters arrive in the wrong order and then `swap` can be used to change the order.

Now that the order is mentioned: the addition operator, introduced in chapter 1 is supposed to have the commutative property: the order does not matter. But it does, as the following examples show:

```
‘A 32 + ‘a =.
32 ‘A +.
```

The first program returns `true`; the second program is met with a runtime error. The commutative property does not apply here. What the runtime error means to say is that adding a character to an integer makes no sense. Now even though ‘A equals 65, it does not mean that they are interchangeable everywhere.

Comparison operators are necessary when comparing floating point values. Oftentimes, floating points look alike when printed in 6 digits, but fail to be the same because the lowest bits are different. In that case, an approximate comparison is needed and that is where the comparison operators come into play, not the equals sign. And that is also why in the test2-directory, floating points are first converted to string and then back to floating point in order to make sure that comparison with `=` becomes possible.

5

Reorder. Reordering data is necessary in a programming language that does not use names when referring to parameters or variables. In addition to `swap`, Joy also has `rolldown`, `rollup`, `rotate` and dipped variants of all four.

The `swap` itself in the example is explained with the `stack` and the `equal` operator. To start with the last one: in chapter 3 the `=` operator is discussed, that can be used for all data types, except lists. Lists can be compared with `equal`; the elements of the lists are compared with `=`.

The `stack` operator transforms the data area into a list; the top of the data area becomes the first member of the list. As the data area, a LIFO structure, is usually pictured with the top most element on the right and a list is pictured starting from the first element, it looks as if a list contains the reverse of the data area. And in the `swap` example in the tutorial, it looks as if the `swap` did not change anything. The `swap` does what it needs to do and so do the other rearranging operators.

As stated, programming languages that can refer to data with names, do not need these rearrangements. It may seem like a weak point of Joy that it needs them, but there are also advantages. Joy can use operators without mentioning where the operator gets its parameters from: the operator already knows that. That makes formulating algorithms more succinct. This succinctness is lost, however, when parameters are used more than once. In that case simple rearrangement is not good enough. The problem could be solved by a system of generalized shuffle operators. The need for such as system has not been urgent enough to warrant its development.

This `swap` operator is also present in the x87 instruction set. That floating point processor operates on a stack of size 8. The same kind of programming that is needed there is also used in Joy. The floating point unit also has control and status words, something that Joy lacks. The Joy virtual processor uses only the instruction set provided by builtins and has no global flags that control the behaviour of the virtual processor. There are some exceptions.

Joy can be implemented with linked lists. In the case of `swap` two new nodes are created and filled with the correct contents; the next pointer of the second node skips the two original nodes. When the data area is implemented as an array, no new nodes are allocated. Instead, the contents of the existing nodes are swapped. It should be stressed that within conditions, this is not really possible, as it prevents restoring the old data area. Here, copies need to be made first.

6

True. The truth values `true`, `false` and the operators `or`, `and`, and `not` can all be discussed together. Encoding of the values requires 1 bit. A value is either `true` or `false`. More logical operators are possible, but `or`, `and`, and `not` are sufficient to build the others. In fact, only one of `nand` and `nor` is needed to construct the rest.

Truth values can be used to evaluate statements in the propositional calculus. Unlike arithmetic, this calculus is complete, consistent, and decidable. That last one can be achieved with truth tables or semantic tableaux.

When linking truth values with the real world, there are some problems. An example is environment variables, such as the HOME directory, where Joy searches usrlib.joy if it was not found in the current directory. That variable may not be present, or it may be empty, or it contains a directory. That is three possible outcomes, instead of two. HOME is not a truth value, but the example will be similar if it was.

A similar problem occurs when extending a database table with a new column. That column will be empty, or in SQL parlance, contain NULL values. Thus, even if the new column is destined to contain a truth value, there are actually three possible outcomes: NULL, `true`, `false`.

Likewise, variables of type truth value may not have been initialized. Or they have been initialized and then the value will be either `true` or `false`. Again, there are three possible outcomes. Now, if the variable is 1 bit wide, it can only store one of two values and if these values are taken to be `true` or `false`, there will be a value. Yes, but if the variable has not been initialized, the outcome is unreliable.

Another example is given by questions. Some questions require a `yes` or `no` answer. But there are always other answers: don't know, don't care, not available, not applicable. Truth values are nice decision makers in a computer program, they are not very useful in everyday life.

Implementing `not`, `and`, and `or` with `nand`. It does require the use of `dup` that is introduced in the second chapter of the tutorial

```
not == dup nand
and == nand dup nand
or == dup nand swap dup nand nand
```