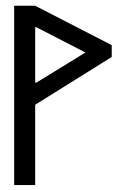
The Joy of Programming of Joy

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



0

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: an Examples 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 Examples 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 Examples 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.

1

TUTORIAL

1.1 Operators

1.1.1 Examples

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

(* Multiplication of signed numbers *)
2 3 * 6 =.

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

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

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

(* Truth value *)
true.

(* Truth value, testing not *)
false not.

(* Test or *)
false true or.

(* Test and *)
true true and.

(* Test not *)
false not.
```

Notes Operators `stack` and `equal` are introduced later on. They are needed here to show the effect of `swap`.

1.1.2 Builtins

- $[\![+ \ldots]\!] \ldots$ M I $\rightarrow [\![\ldots]\!] \ldots$ N
  Numeric N is the result of adding integer I to numeric M. Also supports float.
- $[\![* \ldots]\!] \ldots$ I J $\rightarrow [\![\ldots]\!] \ldots$ K
  Integer K is the product of integers I and J. Also supports float.
- $[\![= \ldots]\!] \ldots$ X Y $\rightarrow [\![\ldots]\!] \ldots$ B
  Either both X and Y are numeric or both are strings or symbols. Tests whether X equal to Y. Also supports float.
- $[\![< \ldots]\!] \ldots$ X Y $\rightarrow [\![\ldots]\!] \ldots$ B
  Either both X and Y are numeric or both are strings or symbols. Tests whether X less than Y. Also supports float.
- $[\![$ swap $\ldots]\!] \ldots$ X Y $\rightarrow [\![\ldots]\!] \ldots$ Y X
  Interchanges X and Y on top of the stack.
- $[\![$ true $\ldots]\!] \ldots \rightarrow [\![\ldots]\!] \ldots$ true
  Pushes the value true.
- $[\![$ false $\ldots]\!] \ldots \rightarrow [\![\ldots]\!] \ldots$ false
  Pushes the value false.

- $[\![$ or ... $]\!]$ ... X Y $\rightarrow$ $[\![$ ... $]\!]$ ... Z
  Z is the union of sets X and Y, logical disjunction for truth values.
- $[\![$ and ... $]\!]$ ... X Y $\rightarrow$ $[\![$ ... $]\!]$ ... Z
  Z is the intersection of sets X and Y, logical conjunction for truth values.
- $[\![$ not ... $]\!]$ ... X $\rightarrow$ $[\![$ ... $]\!]$ ... Y
  Y is the complement of set X, logical negation for truth values.

1.1.3 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.

1.2 Combinators

1.2.1 Examples

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

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

(* Swap elements below the top of the data area *)
1 2 3 swapd stack [3 1 2] equal.

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

(* Evaluate program that was set aside on the data area *)
2 3 [+] i 5 =.

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

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

(* Remove a value from the data area *)
1 2 pop 1 =.

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

1.2.2 Builtins

- ⟦ step ... ⟧ ... A [P] → ⟦ A_1 P A_2 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.

Notes Builtin `nothing` is obsolete. It was used in the definition of the `null` predicate: `null == first nothing sametype`, at the same time making `uncons` a total function. As it is now, `uncons` and also `unswons`, `first`, and `rest` are only defined for non-null aggregates.

The formal notation that is used in `step` and other combinators that are presented later on is similar to the workings of `Moy` and `Foy`. `Joy` and `joy1` adhere to the same semantics but the execution uses the call stack to store part of the continuation. As these four implementations implement the same language, the formalism can be used to describe all of them.

1.2.3 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

Financial accounting is set to "maybe" for two reasons. First of all, the user interface of G3, described in chapters A-Z, is outdated and may not be acceptable for modern usage. And secondly, G3 did not use Joy after all. It used `42minjoy`, a substandard version of Joy.

1.3 Input and output

1.3.1 Examples

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

(* Output a character *)
'A put.

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

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

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

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

(* Reduce an aggregate to a scalar *)
[1 2 3] 0 [+] fold 6 =.

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

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

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

Notes Chapters 1-3 follow the tutorial of 42minjoy. The operators that are introduced are explained in more detail in the Notes, chapters 1-9.

1.3.2 Builtins

- ⟦ of ... ⟧ ... I A → ⟦ ... ⟧ ... X
  X (= A[I]) is the I-th member of aggregate A.
- ⟦ 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.
- ⟦ fold ... ⟧ ... A V_0 [P] → ⟦ A_1 P A_2 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.
- ⟦ 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].

1.3.3 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	"Compare" mitigates a problem in grmtst.joy

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

1.4 Tests

1.4.1 Examples

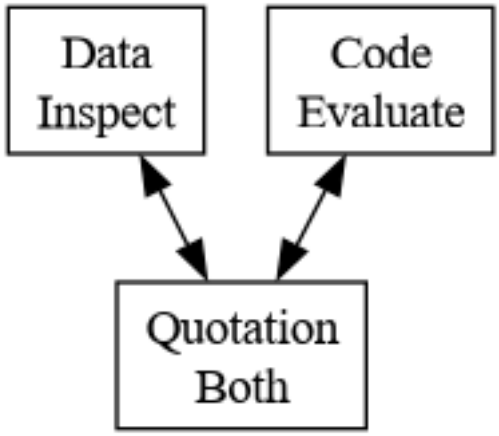
```
(* Output a character as such *)
'A putch.

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

(* Test order of characters *)
'B 'A >.

(* Compare two numbers *)
3 2 >=.

(* Compare two characters *)
'A 'B <=.

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

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

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

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

(* Extract the first element of a non-empty aggregate *)
[1 2 3] first 1 =.
```

1.4.2 Builtins

- ⟦ putch ... ⟧ ... N → ⟦ ... ⟧ ...
  [IMPURE] N : numeric, writes character whose ASCII is N.
- ⟦ getch ... ⟧ ... → ⟦ ... ⟧ ... N
  [OBSOLETE] Reads a character from input and puts it onto stack.

- [> ...] ... 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.

1.4.3 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 is dropped onto the data area where it can be inspected. The primary combinator is `i`: it takes a quotation from the data area and pushes its contents onto 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 and the data area is 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?

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 on 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 on 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.

1.5 Aggregates

1.5.1 Examples

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

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

(* Use the predecessor function to discover the previous char *)
'A pred '@ =.

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

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

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

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

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

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

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

Notes Chapters 4-5 take examples from 42minjoy.lib

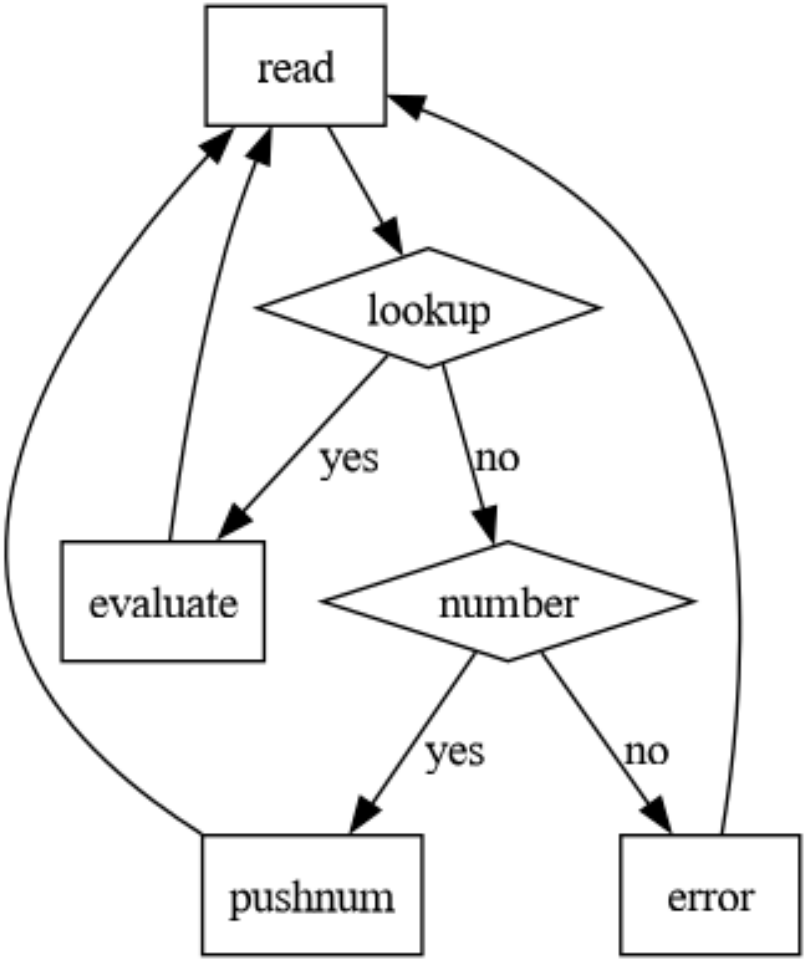
1.5.2 Builtins

- ⟦ branch ... ⟧ ... true [T] [F] → ⟦ T... ⟧ ...
  ⟦ branch ... ⟧ ... false [T] [F] → ⟦ F... ⟧ ...
  If B is true, then executes T else executes F.

- ⟦ 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_1 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.

1.5.3 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.


read
lookup
yes
no
evaluate
number
yes
no
pushnum
error

1.6 Repetition

1.6.1 Examples

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

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

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

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

(* Execute a program while a condition is true *)
10 [] [pop 0 >] [[dup [pred] dip] dip cons] while
[1 2 3 4 5 6 7 8 9 10] equal.

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

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

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

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

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

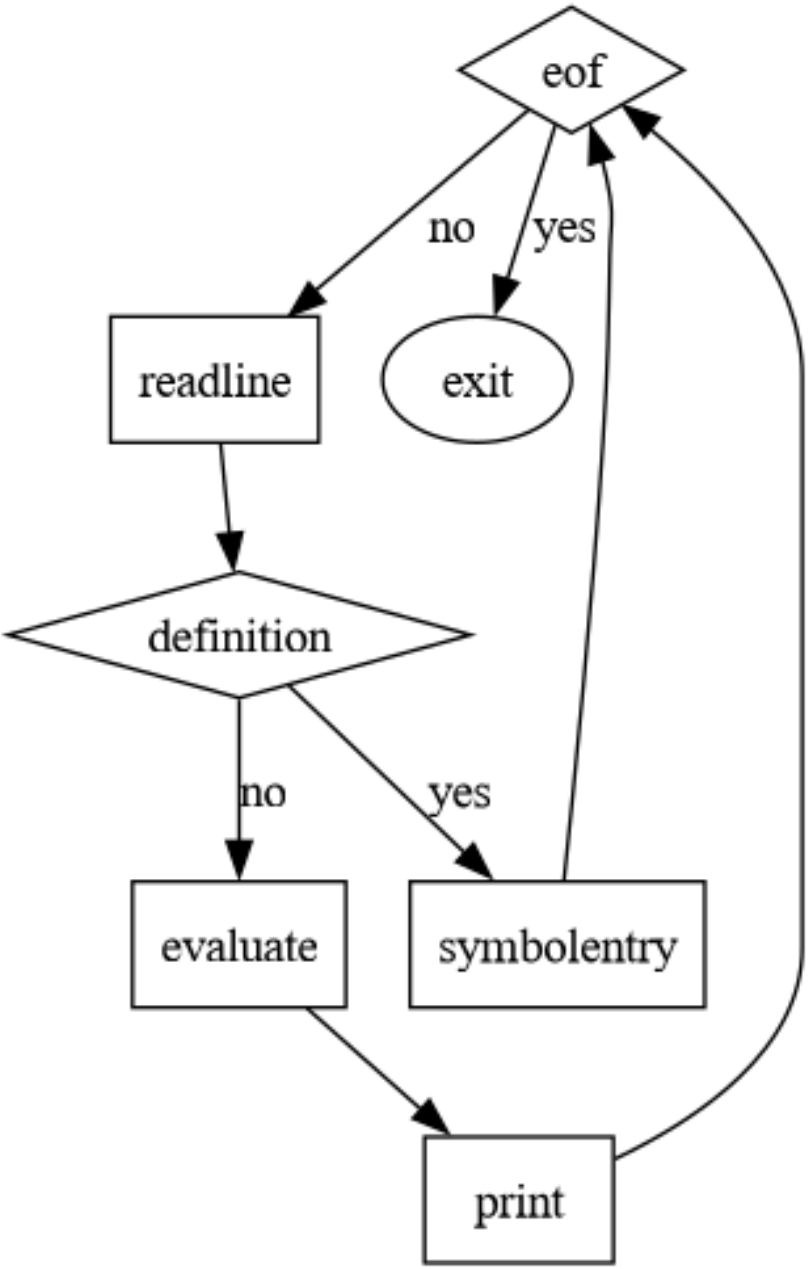
Notes Joy is good in sequential evaluation. Builtin `while` adds iteration, satisfying the requirements for structered programming. Builtin `unary2` is a SIMD instruction (Single Instruction Multiple Data), whereas `cleave` is a MISD instruction (Multiple Instructions Single Data).

1.6.2 Builtins

- ⟦ small ... ⟧ ... X → ⟦ ... ⟧ ... B
  Tests whether aggregate X has 0 or 1 members, or numeric 0 or 1.
- ⟦ 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] [D] → ⟦ B jump-false(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
- ⟦ 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

1.6.3 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.


eof
no
yes
readline
exit
definition
no
yes
evaluate
symbolentry
print

1.7 Recursion

1.7.1 Examples

```
(* Include another file and process that one first *)
"__dump.joy" include.

(* Push the standard input file descriptor *)
stdin file.

(* Open a file for reading *)
"fopen.joy" "r" fopen null not.

(* Read a character from the given file descriptor *)
"fgetch.joy" "r" fopen fgetch '( =.

(* Test whether the end of file has been reached *)
"feof.joy" "r" fopen feof false =.

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

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

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

(* Basic if-then-else example *)
1 [0 >] [true] [false] ifte.

(* Calculate the factorial function using linrec *)
5 [0 =] [succ] [dup pred] [*] linrec 120 =.
```

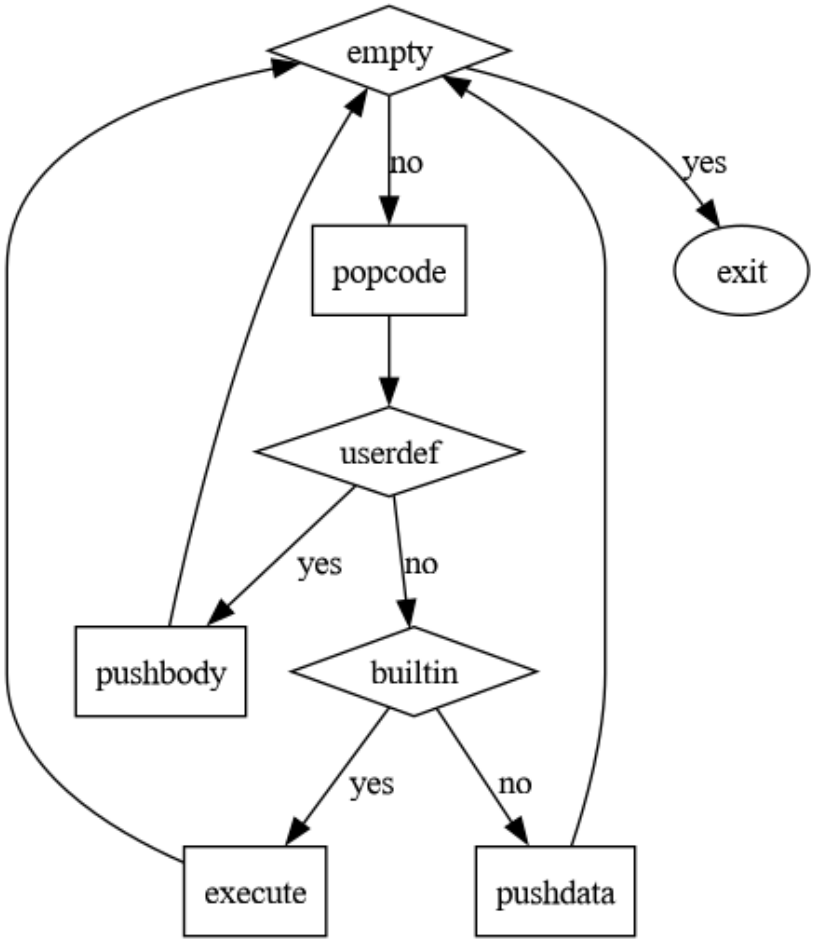
1.7.2 Builtins

- ⟦ include ... ⟧ ... "filnam.ext" → ⟦ ... ⟧ ...
  Transfers input to file whose name is "filnam.ext". On end-of-file returns to previous input file.
- ⟦ stdin ... ⟧ ... → ⟦ ... ⟧ ... S
  [FOREIGN] Pushes the standard input stream.

- ⟦ 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.
- ⟦ 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.
- ⟦ 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-false(1) T jump(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-false(1) T jump(2) (1) F (2) ... ⟧ ...
  Executes B. If that yields true, then executes T else executes F.
- ⟦ linrec ... ⟧ ... [P] [T] [R1] [R2] → ⟦ P jump-false(1) T jump(2) (1) R1 [P] [T] [R1] [R2] linrec R2 (2) ... ⟧ ...
  Executes P. If that yields true, executes T. Else executes R1, recurses, executes R2.

1.7.3 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 area. The inner interpreter is finished when the code stack becomes empty.



1.8 Datatype tests

1.8.1 Examples

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

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

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

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

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

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

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

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

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

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

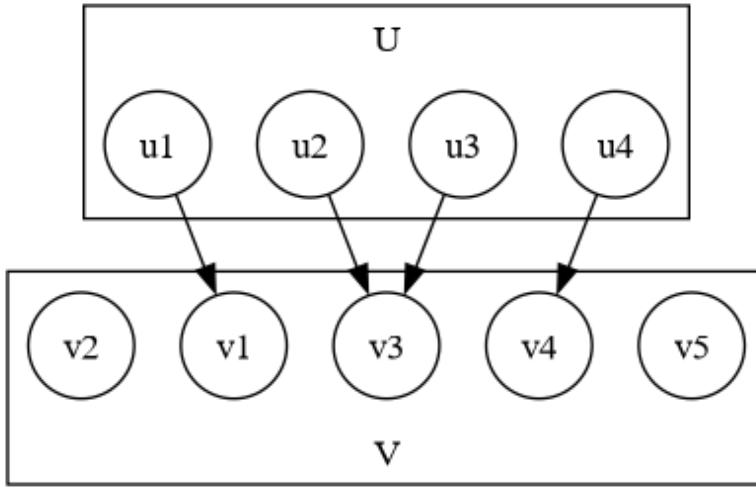
Notes Each of the datatypes mentioned in the Builtins section has its own way of expressing nothing. All nothings compare equal, even though they have a different datatype. There is no constant that can express FILE:NULL.

1	false.
2	'\000.
3	0
4	{}.
5	“”.
6	[].
7	0.0.

1.8.2 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.

1.8.3 Theory



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 $\rightarrow$ V. There is no function from V to U.

1.9 Conditional execution

1.9.1 Examples

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

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

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

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

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

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

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

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

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

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

1.9.2 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.
- ⟦ putchars ... ⟧ ... "abc..." → ⟦ ... ⟧ ...
  [IMPURE] Writes abc.. (without quotes)
- ⟦ split ... ⟧ ... A [B] → ⟦ A_1 B ... A_n B ... ⟧ ... A1 A2
  Uses test B to split aggregate A into sametype aggregates A1 and A2.

1.9.3 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.

1.10 File functions

1.10.1 Examples

```
(* Read a number of bytes from an open file *)
"fread.joy" "r" fopen 7 fread [40 42 32 82 101 97 100] equal.

(* Seek forward in an open file *)
"fseek.joy" "r" fopen 7 0 fseek null.

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

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

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

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

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

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

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

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

1.10.2 Builtins

- ⟦ 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.
- ⟦ 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.
- ⟦ 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.
- ⟦ 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.

1.10.3 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. It 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

Builtin	Runtime	Foreign	Impure
___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
fputstring	Yes	Yes	No
fputchars	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

Builtin	Runtime	Foreign	Impure
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.

If the compiled program is not linked with `scan.c` then the following builtins cannot be used:

```
finclude, get, include, intern
```

1.11 Manual functions

1.11.1 Examples

```
(* Display user defined functions, builtins, and datatypes *)
help.

(* Display concise information about each member of a list *)
[stdin 3.14 [] "" {} 10 'A false maxint helpdetail sum dummy]
helpdetail.

(* Display the manual *)
manual.

(* Display undefined functions *)
undefs.

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

(* Display the current fill of the symbol table *)
__symtabindex.

(* Display hidden symbols *)
_help.

(* Output the manual in html format *)
__html_manual.

(* Output the manual in latex format *)
__latex_manual.

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

1.11.2 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 ...] ... → [...] ... L
  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 ...] ... → [...] ...
  [IMPURE] Lists all hidden symbols in library and then all hidden builtin symbols.
- [___html_manual ...] ... → [...] ...
  [IMPURE] Writes this manual of all Joy primitives to output file in HTML style.
- [___latex_manual ...] ... → [...] ...
  [IMPURE] Writes this manual of all Joy primitives to output file in Latex style but without the head and tail.
- [___manual_list ...] ... → [...] ... L
  Pushes a list L of lists (one per operator) of three documentation strings.

1.11.3 Theory

Computer Architecture
Tempfile
Kilo.joy
Joy
C
Shell
OS
Hardware

1.12 Mathematical functions

1.12.1 Examples

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

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

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

(* Calculate the ceil function *)
1.5 ceil 2 =.

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

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

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

(* Calculate the floor function *)
1.5 floor 1 =.

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

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

1.12.2 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.

1.12.3 Theory

Program	Lines
OS	13000
Compiler	4000
Editor	1000
Joy	182

The OS is MINIX 3, 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.

1.13 More mathematical functions

1.13.1 Examples

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

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

(* Calculate the neg function *)
1 neg -1 =.

(* Calculate the sign function *)
1.0 sign 1 =.

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

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

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

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

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

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

1.13.2 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.

1.13.3 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, N, P, U. 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.

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 area. Primed functions are unprimed when pushed on the data area. 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.

khashl.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.

module.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.

print.c This file implements the P of REPL, according to the autoput settings.

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 area to the code stack. All other functions are straightforward and avoid duplication in the builtins.

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 data area 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 call 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.

1.14 Even more mathematical functions

1.14.1 Examples

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

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

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

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

(* Calculate the ldexp function *)
1.5 2 ldexp 6 =.

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

(* Calculate the round function *)
1.5 round 2 =.

(* Calculate the max function *)
'A 'B max 'B =.

(* Calculate the min function *)
'A 'B min 'A =.

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

1.14.2 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.

1.14.3 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`.

error.c This file is similar to `xerr.c` in Moy, except that there is some additional code that is needed by the compiler. Yes, the compiler spings into action when normally the interpreter would issue an error.

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, are 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.

macros.h This file is similar to, but not identical to the file that is used in Moy.

main.c Compared to Moy, this `main.c` contains more functionality. The symbol table is explained at the start of the file by a posting from Manfred in 2006.

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

optable.c This file is similar to the file `otab.c` in Moy. It contains an extra column on behalf of the compiler.

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

repl.c This file is similar in purpose to the file `repl.c` in Moy. It contains code that is used in Joy, not joy1, and also some code that is used by the byte code compiler.

runtime.h All parameters checks are centralized in this header file. It is similar in purpose to `parm.c` in Moy, not identical. It also offers the opportunity to disable all checks.

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.

symbol.c This file implements the symbol table. This code used to reside in `main.c`.

undefs.c This file contains code that hides inner modules as soon as the outermost module was processed.

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 was slower than the one in the legacy version, because it used a non-recursive algorithm during the collection.

writ.c This file is similar to the one used in Moy.

1.15 Shufflers

1.15.1 Examples

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

(* Identity program, does nothing *)
id.

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

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

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

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

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

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

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

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

1.15.2 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

1.15.3 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 data area, but this old version expects a number on top. 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 data area, the whole data area, or only a newline. If there is something on the data area, 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 data area. 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 data area 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 data area. 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.

1.16 Settings

1.16.1 Examples

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

(* Get the setting of the autoput flag *)
autoput 1 =.

(* Get the setting of the undeferror flag *)
undeferror 1 =.

(* Get the setting of the echo flag *)
echo 0 =.

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

(* Get the timestamp *)
time.

(* Get a pseudo random number *)
rand.

(* Push the standard output file descriptor *)
stdout file.

(* Push the standard error output file descriptor *)
stderr file.

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

1.16.2 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.

1.16.3 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.

1.17 More settings

1.17.1 Examples

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

(* Issue a system command *)
"ls true.joy" system.

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

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

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

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

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

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

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

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

1.17.2 Builtins

- ⟦ intern ... ⟧ ... "sym" → ⟦ ... ⟧ ... sym
  Pushes the item whose name is "sym".
- ⟦ 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 ... ⟧ ... → ⟦ ... ⟧ ...
  [IMPURE] 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).
- ⟦ chr ... ⟧ ... I → ⟦ ... ⟧ ... C
  C is the character whose Ascii value is integer I (or logical or character).

1.17.3 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 checker.

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.

1.18 Selections

1.18.1 Examples

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

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

(* Tell whether some elements satisfy a condition *)
[1 2 3] [2 <] some.

(* Tell whether all elements satisfy a condition *)
[1 2 3] [4 <] all.

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

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

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

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

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

1.18.2 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_1 B if-true-jump(1) ... A_n B (1) ... ⟧ ... X
  Applies test B to members of aggregate A, X = true if some pass.
- ⟦ all ... ⟧ ... A [B] → ⟦ A_1 B if-false-jump(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 <=, =, >=.

1.18.3 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 coverage 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.

```
Current view: top level              Coverage Total Hit
        Test: coverage.info     Lines: 94.0 % 4495 4225
   Test Date: 2024-12-05    Functions: 92.4 %  356  329

                Line Coverage      Function Coverage
  Directory   Rate    Total Hit    Rate    Total Hit
  Moy          85.3 % 1842  1572    75.9 % 112    85
  Moy/src     100.0 % 2653  2653   100.0 % 244   244
```

The src-directory should display 100%. The way to generate this report is by executing the following script:

```
lcov -d . --zerocounters
cmake -G "Unix Makefiles" .
cmake --build .
if [ $? -eq 0 ]
then
lcov --capture -b . -d . -o coverage.info
genhtml --prefix "C:" coverage.info
fi
```

Coverage 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.

1.19 Tree functions

1.19.1 Examples

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

(* 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.

(* 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 =.

(* Use tailrec when building a list of integers *)
[] 10 [0 =] [pop] [dup [swons] dip pred] tailrec
[1 2 3 4 5 6 7 8 9 10] equal.

(* Read a line from a file *)
"fgets.joy" "r" fopen fgets "(* Read a line from a file *)\n" =.

(* 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.

(* 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.

(* 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.

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

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

1.19.2 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.

1.19.3 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 are 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`.

1.20 Internal functions

1.20.1 Examples

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

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

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

10 g-fib 55 =.

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

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

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

(* Report the current size of memory allocation *)
__memoryindex.

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

(* Set the tracegc flag *)
0 __settracegc.

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

1.20.2 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.

1.20.3 Maintenance

	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.

1.21 More file functions

1.21.1 Examples

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

(* Flush the stdout stream *)
stdout fflush stdout =.

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

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

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

(* Write a string to a file *)
"test" "w" fopen "test" fputchars.
$ rm test

(* Report the position in a file *)
"ftell.joy" "r" fopen   # fp
0 2 fseek pop           # fp, removing success condition
ftell                   # fp offset
149 =.                  # offset

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

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

(* Execute the app12 function *)
1 2 3 4 [+] app12 stack [6 5 1] equal.
```

1.21.2 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.

1.21.3 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.

1.22 Time functions

1.22.1 Examples

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

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

(* Get the local time *)
time localtime 6 take.

(* Get the Greenwich mean time *)
time gmtime 6 take.

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

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

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

(* Write a string to a file *)
"test" "w" fopen "test" fputstring.
$ rm test

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

(* Change the type of an item *)
"Hello, World" 12 casting.
```

1.22.2 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.

1.22.3 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 data area 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 data area. 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: Local definitions cannot call each other. This has been repaired.

There is also the possibility to define modules within modules, a consequence of how they are processed along the lines of `HIDE .. IN .. END`. It is not part of the design, not shown in `modtst.joy` and not mentioned in the user manual. It is supported now, but without guarantees.

1.23 Copy functions

1.23.1 Examples

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

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

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

1.23.2 Builtins

- ⟦ ord ... ⟧ ... C → ⟦ ... ⟧ ... I
  Integer I is the Ascii value of character C (or logical or integer).
- ⟦ 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.

1.23.3 Theory

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 λ abstraction.

constant ::= dup | swap | 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 ::= dup | swap | pop ...

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

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

1.24 Imperative functions

1.24.1 Examples

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

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

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

1.24.2 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.

1.24.3 Theory

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.

2

G3 User Manual

A

```
------------------------------------------------------------
ELYA KAPSALON                BOEKHOUDEN                  G3
------------------------------------------------------------

                              versie 1.1

                      Copyright Saru Janpu 2020

                    Grootboek Gemakkelijk Gemaakt

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 left 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 right is the name of the program. The middle of the header mentions what this screen is

about. In the lower left there is one of the two clues about what keys can be pressed: how to continue and how to leave.

```
------------------------------------------------------------
ELYA KAPSALON                ACCOUNTING                   G3
------------------------------------------------------------

                            version 1.1

                    Copyright Saru Janpu 2020

                  Financial Accounting Made Easy

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 left 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. The box at the top of the screen 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

```
------------------------------------------------------------
SARU JANPU                  MAIN MENU                    G3
------------------------------------------------------------

                            1. register
                            2. settings
                            3. 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 menus 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.

C

```
------------------------------------------------------------
SARU JANPU             REGISTER                          G3
------------------------------------------------------------

                       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 real world references to be added 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

```
------------------------------------------------------------
SARU JANPU              SETTINGS                        G3
------------------------------------------------------------

                        1. create account
                        2. standard accounts
                        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

```
------------------------------------------------------------
SARU JANPU              REPORTS                         G3
------------------------------------------------------------

                        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

```
------------------------------------------------------------
SARU JANPU                          CASH                 G3
------------------------------------------------------------

cash          : [0                  ]
date          :
amount        :  0,00
account       :  0
VAT amount    :  0,00

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.

```
------------------------------------------------------------
SARU JANPU                          CASH                 G3
------------------------------------------------------------

date          : [      ]
amount        :  0,00
account       :  0
VAT amount    :  0,00

Your Choice   :
```

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

G

```
------------------------------------------------------------
SARU JANPU                          BANK                  G3
------------------------------------------------------------

bank          : [0                  ]
date          :
amount        :  0,00
account       :  0
VAT amount    :  0,00

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.

```
------------------------------------------------------------
SARU JANPU                          BANK                  G3
------------------------------------------------------------

date          : [      ]
amount        :  0,00
account       :  0
VAT amount    :  0,00

Your Choice   :
```

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

H

```
------------------------------------------------------------
SARU JANPU          GENERAL TRANSACTION                  G3
------------------------------------------------------------

date         : [      ]
1st account  :  0
amount       :  0,00
2nd account  :  0

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 "1st account" field is the receiving account of the amount that is entered in the next field. The "2nd account" 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.

I

```
------------------------------------------------------------
SARU JANPU                          NOTES                G3
------------------------------------------------------------

seqnr        : [                    ]
date         :
amount       :  0,00
account      :  0
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

```
------------------------------------------------------------
SARU JANPU              YEAR END CLOSING                G3
------------------------------------------------------------

first date   : [      ]          last date    :
date         :
amount       :  0,00
account      :  0
notes        :

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

```
------------------------------------------------------------
SARU JANPU                      CREATE ACCOUNT          G3
------------------------------------------------------------

account      : [                  ]
description  :
amount 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

```
------------------------------------------------------------
SARU JANPU              STANDARD ACCOUNT                G3
------------------------------------------------------------

description  : [                              ]
account      :  0

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

```
------------------------------------------------------------
SARU JANPU              PERCENTAGES                     G3
------------------------------------------------------------

VAT code     : [                              ]
VAT %        :  0,00

Your Choice  :
```

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

N

```
        CHART OF ACCOUNTS
        ******
        051224


1000    b               Cash
```

```
1010    b               Bank
1672    b               Debt to Owner
4000    w               KvK
9019    w               Result 2019
~
~
```

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

```
        STANDARD ACCOUNTS
        ******
        051224


Bank    1010
Cash    1000
~
~
```

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

P

```
        PERCENTAGES
        ******
        051224


~
~
```

This screen should present the VAT percentages.

Q

```
------------------------------------------------------------
SARU JANPU            TRANSACTION JOURNAL               G3
------------------------------------------------------------

first date   : [      ]          last date    :
first seqnr  :  0                last seqnr   :  0

Your Choice  :
```

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

```
        TRANSACTION JOURNAL
        ******
        051224


1       1000    281019                        50,00
1       1672    281019                       -50,00
2       1000    281019                       -50,00
2       4000    281019                        50,00
3       4000    311219                       -50,00
3       9019    311219                        50,00
4       1672    311219                        50,00
4       9019    311219                       -50,00
~
~
```

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

R

```
------------------------------------------------------------
SARU JANPU                LEDGER VIEW                   G3
------------------------------------------------------------

first date   : [      ]          last date    :
first seqnr  :  0                last seqnr   :  0
account      :  9019             Result 2019
```

```
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.

```
        LEDGER VIEW
        ******
        051224


3       9019    311219                      50,00
4       9019    311219                     -50,00
                        --------------------------
        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

```
------------------------------------------------------------
SARU JANPU               PROFIT & LOSS                   G3
------------------------------------------------------------

first date   : [      ]          last date    :
first seqnr  :  0                last seqnr   :  0

Your Choice  :
```

This is the selection screen of the Profit & Loss statement. For the current year it should make 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
        ******
        051224


                          --------------------------
        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

```
------------------------------------------------------------
SARU JANPU                BALANCE SHEET                  G3
------------------------------------------------------------

first date   : [      ]          last date    :
first seqnr  :  0                last seqnr   :  0

Your Choice  :
```

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

```
        BALANCE SHEET
        ******
        051224


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

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

U

Before doing postings, it is necessary to setup a new directory where these postings can be stored. The install program installs G3 with a working directory in C:\ProgramData\G3. That is a hidden directory. The shortcut on the desktop is stored in C:\Users\Public\Desktop. This shortcut needs to be copied to the desktop of the user that starts G3, where the shortcut can be modified.

The properties page of the shortcut shows the program name, G3, and the working directory, C:\ProgramData\G3.

On the first run, the screen size is way too large. The program has been designed for a 80x25 display. This can be set in the shortcut. Also, after the first run, when lang.joy has been copied to the working directory, instead of calling G3, it is better to call Joy with parameters "g3.joy." That takes less memory than the compiled batch file, G3.

The memory use after modifying the shortcut is:

```
P joy (32-bits) (2)    7.4 MB
  P joy (32-bits)      0.9 MB
  C Consolewindowhost  6.5 MB
```

The options tab with checkboxes: all of them can be cleared and buffers nullified. The buffers will be reset to 1 by the system. The screen size should be set to 80x25.

There is a separate INSTALL.pdf where all of this is explained with screen shots.

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
        ******
        051224


2000    km              Odometer 74-PH-KS
2001    km              Work related travel
2003    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.

```
        LEDGER VIEW
        ******
        051224


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
        ******
        051224


2000    km              Odometer 74-PH-KS
2001    km              Work related travel
2003    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.

```
        LEDGER VIEW
        ******
        051224
```

```
4       3000    010115                      160,00
5       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
        ******
        051224


1000    b               Cash
1491    b               VAT to be paid
2000    km              Odometer 74-PH-KS
2001    km              Work related travel
2003    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
8000    w       1       Sales low VAT
~
~
```

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:

Account	Description
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:

```
        STANDARD ACCOUNTS
        ******
        051224


1       1491
Cash    1000
~
~
```

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
        ******
        051224


1       900
~
~
```

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

```
        PROFIT & LOSS
        ******
        051224
```

```
8000    Sales low VAT                       -9,17
                    --------------------------
        Net Profit                          -9,17
~
~
```

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

```
        BALANCE SHEET
        ******
        051224


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

X

```
        CHART OF ACCOUNTS
        ******
        051224


1000    b               Cash
1491    b               VAT to be paid
1672    b               Own Debt
2000    km              Odometer 74-PH-KS
2001    km              Work related travel
2003    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
4000    w               Travel costs
```

```
8000    w        1      Sales low VAT
9015    w               Result 2015
~
~
```

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

```
        TRANSACTION JOURNAL
        ******
        051224


6       1000    010215              10,00
6       8000    010215              -9,17
6       1491    010215              -0,83
7       4000    010215              14,44 38 km*2*19 cent/km
7       1672    010215             -14,44 38 km*2*19 cent/km
~
~
```

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

```
        PROFIT & LOSS
        ******
        051224


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


        BALANCE SHEET
        ******
        051224


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

```
                          -------------------------
        Net Profit                           -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.

```
        LEDGER VIEW
        ******
        051224


8       9015    311215                        5,27
11      9015    311215                       -5,27
                        -------------------------
        Balance                               0,00
~
~
```

This summarizes the year 2015. Posting number 8 shows a loss of 5 EUR 27 cents.

Y

```
        TRANSACTION JOURNAL
        ******
        051224


6       1000    010215                10,00
6       8000    010215                -9,17
6       1491    010215                -0,83
7       4000    010215                14,44 38 km*2*19 cent/km
7       1672    010215               -14,44 38 km*2*19 cent/km
8       9015    311215                 5,27
```

```
8       4000    311215              -14,44
8       8000    311215                9,17
9       1000    311215              -10,00
9       1672    311215               10,00
10      1491    311215                0,83
10      1672    311215               -0,83
11      9015    311215               -5,27
11      1672    311215                5,27
~
~
```

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.

```
7       "38 km*2*19 cent/km"
~
~
```

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 menus 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.

Installation

G3 is best installed with the installer program. Windows only. The installer does not install the source code of G3. What it does install:

```
LIB
g3.bat              1 kB
g3.boc             15 kB
g3                  3 kB
g3                  1 kB
joy.exe           199 kB
Uninstall.exe     108 kB
usrlib.joy          2 kB
```

The LIB directory contains the library files, used by Joy. G3 does not use them. g3.bat is the program that starts joy.exe. The g3 of 3 kB is the shortcut that is copied to the desktop by the installation program. The second g3 is a link to the website. Then there is joy.exe, the full language processor. Uninstall.exe removes these files, as well as the shortcut on the desktop and the directory where these files are located, G3-JOY-1. It does not delete the parent directory, Saru Janpu. The usrlib.joy contains the first code that is evaluated by joy.exe, when joy.exe is started on its own.

What happens when the user clicks on the shortcut on the desktop is that the program cmd.exe is asked to create a terminal screen and start g3.bat. g3.bat asks joy.exe to start processing g3.boc. And then the splash screen of G3 is presented. The user is asked to press the right arrow key.

When the user leaves the program by pressing the left arrow key, the batch file takes over again and asks cmd.exe to exit. The terminal screen then disappears.

In addition to the shortcut on the desktop, also a link in the start menu is created. This starts joy.exe for anyone who is curious about Joy. g3.boc is compiled Joy source code.

The explanation in chapter U is no longer valid. There is no language file. The texts on the screen are embedded in g3.boc. The language is Dutch. The company name in the upper left is the company that uses G3.

Running

The installer program also creates a working directory where the data files of G3 are stored. The g3.bat is copied to that directory. When running G3 the following files will be present:

```
account.txt
g3.bat
journal.txt
percent.txt
stdacct.txt
textblk.txt
```

When the posting presented in the chapters A-Z are added to these files, they will all be 1 kB in size. Because of the .txt extension, it is easy to look at these files with notepad. They should not be modified directly. Not by the owner of the company and certainly not be a bookkeeper, accountant, controller, consultant, inspector. The only ligitimate way to modify these files is with G3.

That said, the owner of the company, who is also owner of the data and responsible for reports that are generated from that data, can theoretically modify the data directly. The creator of G3 cannot forbid such action.

3

Notes

1

Addition. This is introduced in paragraph 1.1.1, first example. The comments mention signed numbers. Numbers are taken from $\mathbb{Z}$ and that set already has signed numbers, so why is it mentioned explicitly 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 worth-

while. 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 15^{th} 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. This is introduced in paragraph 1.1.1, second example. 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 introduced in paragraph 1.1.1, third example. 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. This is introduced in paragraph 1.1.1, fourth example. 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. This is introduced in paragraph 1.1.1, fifth example. 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. This is introduced in paragraph 1.1.1, sixth example. 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.

Yet another example is the statement: "Windows uses \ to separate paths." Is this statement true or false? It is certainly not false, because sometimes Windows needs \ and not /. On the other hand, in most cases / can be used. That means that this statement is somewhere between true and false.

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
```

7

Step. This is introduced in paragraph 1.2.1, first example. Step processes an entire aggregate. That is ok, if the aim is to summarize the contents, as in the example given. But what if only part of the aggregate needs to be selected?

The question is how `strchr` can be implemented in Joy. Here is a solution that is vector-friendly:

```
HIDE
  _ == [dup null] [] [[over over first =] [] [rest _] ifte]
       ifte
IN
  strchr == swap _ popd
END
```

The approach is non-idiomatic. It makes use of direct recursion, whereas it is preferable to use one of the combinators that encapsulate recursion, allowing names to be omitted. Here, then is an idiomatic solution:

```
LIBRA
  strchr == swap [[null] [first =] sequor not] [rest] while
            popd.
```

This solution makes use of a complicated condition that is not vector-friendly. It is too bad that the two solutions differ. Ideally, the source should remain the same, regardless of the underlying implementation.

What this means is that the vector-based implementations, although faster, are not the preferred implementations, because they require more effort on the part of the programmer to make them work correctly and efficiently.

Step can also be used to reverse an aggregate in a vector-friendly way:

```
[] step stack
```

This works, because `step` processes an aggregate from left to right and `stack` uses the top most element as the first element of the new aggregate. But the above only works on an empty data area and afterwards this area must be cleaned from all elements that `step` left behind:

```
[[] unstack] dip.
```

8

Cons. This is introduced in paragraph 1.2.1, second example. Cons is used to build lists. One element is added to the front of the list. If a list is implemented with links, this is a cheap operation: simply point the link to the old list. The old list is not affected.

If the list is implemented with a vector, then the story is different. If the vector is stored with the first element at location 0, then a new vector needs to be allocated with an empty slot in front. That slot can then be filled with the new element.

If the vector is kept in reverse, then another option is available: add the new element at the end of the vector where the head of the vector is located. The old vector is not affected, because its header does not see the new element. The new vector receives its own header with an extra element. All of this is done by the function *vec_shallow_copy_take_ownership*. This name is on purpose extra long, because what is done is a bit dangerous, but it seems to work. A vector has an ownership bit in the header, sort of 1-bit reference count, that mentions whether the vector owns the array where the data is stored. The new vector can only add a new element if the old vector owned the array. The function with the long name takes away the ownership from the old vector and gives it to the new one.

As already said, the operation is a little dangerous, but gives good performance. The Moy implemention uses this kind of vector; Foy does not use it. So, a performance comparison between Moy and Foy is possible.

But `cons` is not the only way to construct a vector. Data can also be spread out on the data area, where it can be summarized with the `stack` operator. This may be faster then consing one element at the time. The data area is the only place where destructive updates can be done, without disturbing the functional nature of Joy. And that makes it the fastest location to do manipulations.

There is also the function *vec_shallow_copy* that does an equally dangerous operation. If the size of a vector needs to be reduced, then it is sufficient to make a copy of the header and decrease the number of elements in the new header. This also is a dangerous operation, but again, it seems to work. And again, this function is used in Moy, not in Foy.

Vector based implementations are faster when huge tables need to maintained. Now, Joy is destined to be used for small programs and huge tables are not its target audience. The concern for performance may also be a little old fashioned, considering that PC's are now a factor 1000 faster than they were at the start. And have a million times more memory.

9

Swapd. This is introduced in paragraph 1.2.1, third example. Reordering parameters is a necessary evil consequence of using parameters with no name. This is one among the six builtins that reorder parameters below the top. This particular one is used in the definition of `sum`:

```
sum == 0 [+] fold
```

Fold is also builtin, but can be defined as:

```
fold == swapd step
```

The thing about `sum` is that `+` is a binary operator. Sum is used to sum the numbers in a list and that list provides only one parameter to `+`. That is why the 0 is necessary in the definition of sum. But for the purposes of `step` this parameter is at the wrong location: `step` wants the aggregate on top. That is why the `swapd` is needed. It all fits together, but takes some tinkering to get it right.

The other dipped reordering operators are: `dupd`, `popd`, `rolldownd`, `rollupd`, and `rotated`. The latter three look at 4 values in the data area. For most purposes this is sufficient.

Dip. This is introduced in paragraph 1.2.1, fourth example. Bypassing the top of the data area is possible with `dip`. Whereas Joy is claimed to have no state, an application may have state.

Take for example a text editor. It needs to store the file in a buffer, it needs to be able to locate the (x,y) position of the cursor on the screen, it needs to remember the cursor position in the buffer, and it needs to maintain a flag that tells whether the buffer has been modified or not.

All that state can be stored on top of the data area. That means that any computation that is needed must bypass this state. That can be done with `dip`. There is an alternative and that is to store the state in a variable. This saves a lot of dipping, but is not the functional way of doing things.

What solution is chosen depends on what is most convenient. The language is just a tool to get things done and if that results in a non-functional approach, then such an approach may be acceptable.

I. This is introduced in paragraph 1.2.1, fifth example. Similar to `dip`, except that it does not bypass the top of the data area.

10

bugs. Yes, software has bugs. This seems to be inevitable. Now, if bugs are defined as unexpected failures, it is not possible to compile a list of bugs. Such a list would be a list of expected failures and then they would not be considered bugs. Even so, it might be beneficial to mention a number of such failures. The list that follows is extracted from 42minjoy and G3.

MAXMEM

The full version of Joy doesn't have it, but 42minjoy uses `MAXMEM` to limit the number of nodes that are available. The reason to remove `MAXMEM` from the full version of Joy is the `fread` builtin. This builtin stores every character read in a node. If `MAXMEM` is set to 20000, then files larger than 20K cannot be read. The value of `MAXMEM` can be increased, but whatever value is chosen, there will always be text files that cannot be read, even though the computer may have memory to spare.

What is said about `MAXMEM` can be applied to all variables in the C source that start with `max`.

redefinition

This is not possible in 42minjoy. Definitions must be sorted according to the ASCII alphabet and the system checks that they are in strict increasing order, making redefinition impossible. It also doesn't make sense, as all definitions must be given before any program can execute.

clock

This function may not be available in old C libraries and has been replaced by the `time` function. Timings are given in seconds instead of milliseconds.

#-comments

Single line comments are not available in 42minjoy. They can be added, but there is no urgent need to do so. 42minjoy can be adapted to meet the requirements of an application and need not be prepared for all possible applications.

strftime

This function is not available in old C libraries and if it is, it depends on the C library what format strings are available. What this means is that the language definition lacks precision. The Joy language depends on the C library that is used to compile the binary. It is now possible to give a list of format strings that are safe to use:

Format	Meaning
%a	abbreviated weekday name (e.g., Mon)
%A	full weekday name (e.g. Monday)
%w	weekday number (0 = Sunday)
%d	day of the month (01-31)
%b	abbreviated month name (e.g., Jan)
%B	full month name (e.g., January)
%m	month number (01-12)
%y	year without century (00-99)
%Y	year with century (e.g. 2026)
%H	hour (00-23)
%I	hour (01-12, 12-hour clock)
%p	AM/PM indicator
%M	minutes (00-59)
%S	seconds (00-60, leap second allowed)
%z	UTC offset (e.g., +0100)
%Z	time zone name
%j	day of the year (001-366)
%U	week number (sunday as first day of week)
%W	week number (monday as first day of week)
%c	locale's date and time representation
%x	locale's date representation
%X	locale's time representation
%%	literal percent sign

The strings %z and %Z are not guaranteed to work in Joy. It depends on the C-library that was used to link the Joy binary against. The time structure has fields that are not filled by Joy. No wonder that sometimes the UTC offset is +0000 in Joy. The name of the time zone may be printed correctly.

VAT-amounts

This is a possible failure in G3. Amounts are multiplied by 10000 in the computation of VAT-amounts. That means that only amounts up to 14 digits have their VAT-amount calculated correctly. If VAT must be calculated for larger amounts, it is still possible to manually correct the wrongly calculated amount.

Translations

The interface should be in the local language, adapted to the specific terminology of accounting in that language. This requires changes to `lang.joy`. Fortunately, the end user should be able to make the necessary adjustments.

Files

The files are vulnerable to manual modification. Whereas in the previous version those files were Joy source code and read as such, they are now more or less fixed in their format: tab separated fields, either numeric or string. The string fields are surrounded by double quote characters. If someone modifies these files with a text editor, the possibility exists that the files become broken and unreadable for G3.

11

ACID. G3 uses a database and databases are subjected to the *ACID* test.

Atomicity

Postings consist of 2 lines, or 3 when VAT is involved, or more in the Year End Closure. All lines of these postings should be written or none or them should be written. There is one error condition that could result in a partial write and that is a full hard disk. Now, if anything can happen, it will happen. So, this must be considered a FAIL.

Consistency

Postings can refer to accounts that do not exist. It is possible to add an account later on. Apart from that, consistency is guaranteed. This can be considered a PASS.

Isolation

G3 is a single user system. This guarantees that each transaction is isolated from all other transactions. Even if a user updates the database from two different terminals, the worst that can happen is that two transactions receive the same sequence number. This does not negatively affect the workings of the system. This should be considered a PASS.

Durability

Postings, once written, cannot be changed. This guarantees durability. Except that the owner of the company is also owner of the data in the books. And the owner has access to `journal.txt` and can change the data with an editor. Such a change is outside of the workings of the application. As far as the application is concerned, this can be considered a PASS.

12

bugs, errors, and defects. All software has bugs. Studies show that there is one bug in every 1000 lines of code. That is why note 10 mentions 8 bugs. There are 8000 lines of code in the G3 application: 5000 in C and 3000 in Joy. So, 5 bugs are mentioned in the C code and 3 in the Joy code. But there can be more.

Maxint

Both accounts and amounts allow 19 digits to be entered. But entering 19 nines does not result in 19 nines. It results in 9223372036854775807, because that is the largest amount that can be presented in a 64 bit signed integer. Needless to say, it will not be necessary to maintain account numbers that are that large, nor will amounts in a small company be as large as that.

Full disk

As mentioned in Note 11 a full disk may result in a partial write that needs to be corrected manually. It is possible to add some code that does the correction automatically. Here is a description of such code:

- Before posting, record the current size of the file.
- At the end of the posting, before closing the file, record the error status of the file.
- After posting, inspect the error code. If it is zero, then the code and the old size can be removed. Done.
- If the error code is not zero, record the new size of the file.
- Position the file pointer at the old size.
- Write a number of space characters, equal to the difference of the new size and the old size.
- Remove old size, error code, and new size. Done.

This method works for all files that start with a number. It so happens that the reader does not care how long the lines in the file are and the conversion from text to number does not care how many leading spaces there are.

But is it worth the effort? The G3 application is meant for small companies. The resulting files are also small. And if there is a partial write of a posting, it can be repaired.

13

The Joy repository received a review:

Issue	Severity	Notes
No allocation error checks on non-MSVC	CRITICAL	utils.c:40,210,258,301
No recursion depth limit	CRITICAL	interp.c:77
Memory errors only checked on MSVC	CRITICAL	utils.c multiple
String strdup() not tracked by GC	HIGH	utils.c:343
No thread safety on global state	HIGH	globals.h
Missing argument in abortexecution_()	HIGH	symbol.c:114
Memory leaks on error paths	HIGH	symbol.c:39,119
Recursive mark_ptr risks stack overflow	MEDIUM	gc.c:197
Aggregate operation code duplication	MEDIUM	map/filter/split repeat switch/case logic
Limited recursion depth (5 dump levels)	MEDIUM	Deep recursion fails
File descriptor leaks after longjmp	MEDIUM	scan.c
No C-level unit tests	MEDIUM	Primitives tested only in Joy
Manual test execution required	MEDIUM	Not fully automated in CI
Library setup required	MEDIUM	Test needs ~/usrlib configured

Issue	Severity	Notes
Fixed nesting limit (DISPLAYMAX=10)	LOW	Stack overflow if exceeded
Ghost symbols accumulate in vector	LOW	Memory not reclaimed after module exit
Numeric prefix collision risk	LOW	Hide ID could theoretically collide with symbol names
Deep macro nesting obscures errors	LOW	Difficult debugging
Generic error messages	LOW	Limited context for users
No stack trace in error messages	LOW	error.c
No test coverage metrics	LOW	Cannot measure completeness

13.1

No allocation error checks on non-MSVC.

Yes. This looks odd and should be changed. There is a difference between Linux and Windows: Windows does not overcommit. But overcommit can also be disabled in Linux and in that case malloc can return 0. Must fix.

13.2

No recursion depth limit.

Yes. When a program uses more stack space than available, it is aborted by the OS. The C programmer can set a limit and prevent that the program ends. In both cases there is a problem and the Joy programmer needs to change a recursive solution by an iterative one. This is a problem of all programming languages and was solved in the Moy repository. At a price. Moy is slower than Joy. No fix needed.

13.3

Memory errors only checked on MSVC.

Yes. This is a duplicate of 13.1 Actually there are more occurrences in the src-directory where the return value of malloc and strdup is not tested. If the return value of malloc is tested somewhere, it should be tested everywhere. Must fix.

13.4

String strdup() not tracked by GC.

No. The string is interned by newnode and then released with a call to free. No fix needed.

13.5

No thread safety on global state.

Yes. Joy is single threaded. No fix needed.

13.6

Missing argument in abortexecution_().

Yes. It looks odd when doing a grep abortexecution on all c-files, but this call is within comments and the comment explains why that is. No fix needed.

13.7

Memory leaks on error paths.

No. Memory is allocated in enteratom and that is after a possible error. Memory allocated with GC_strdup precedes the possible error, but that memory will be reclaimed by the garbage collector. There are memory leaks in the symbol table. An example is `verbose` in `usrlib.joy`. The old definition is replaced and not garbage collected. There is no garbage collection of definitions. No fix needed.

13.8

Recursive mark_ptr risks stack overflow.

Yes. Recursion risks stack overflow. This can only happen in very deep data structures and no such structures are used in Joy source code.

The BDW garbage collector does not use recursion during marking. Indeed there exists an algorithm, called pointer reversal, that can do the marking without recursion. Such measures are needed in very general garbage collectors, not in collectors that are designed to support Joy. No fix needed.

13.9

Aggregate operation code duplication.

Yes. There is similarity in code between different builtins and macro's are used when the code is almost the same. These macro's are helpful, as they reduce the number of source files that need to be modified whenever the need arises. But they are not pretty. And conditional compilation within these macro's is not possible. No fix needed.

13.10

Limited recursion depth (5 dump levels).

No. dump1-5 are not used in recursion. They are roots for garbage collection. No fix needed.

13.11

File descriptor leaks after longjmp.

Yes. There is no leak, but the fclose before the longjmp is better placed after the longjmp. As it is now, there is a read from a closed file descriptor. Fortunately, the C library also returns EOF in that case. It is better to read at EOF on an open file. Must fix.

13.12

No C-level unit tests: Primitives tested only via Joy.

Yes. In fact, there are no tests at all. The test-directories are only used to achieve 100% coverage. Testing is a bit more involved that the examples given. No fix needed.

13.13

Manual test execution required: Not fully automated in CI.

Yes. It is not manual, there is a shell script that executes the tests. There is no CI that does the tests. CI is not what it used to be and is a bit over an overkill for a one-person repository. No fix needed.

13.14

Library setup required: Tests need ~/usrlib configured.

Yes. It is convenient that Joy programs have access to the library from every directory where they are executed. This requires some effort. No fix needed.

13.15

Fixed nesting limit (DISPLAYMAX=10): Stack overflow if exceeded.

Yes. If HIDE or MODULE is nested more than the DISPLAYMAX, the Joy programmer is overdoing it. The same applies to INPSTACKMAX. No fix needed.

13.16

Ghost symbols accumulate in vector: Memory not reclaimed after module exit.

Yes. Symbols in the HIDE section of a module, are no longer accessable when the module is finished, or rather that is how it should be. It is true for the outermost module. Even so, these symbols must remain in the symbol table: there is an option -s to print the symbol table. No fix needed.

13.17

Numeric prefix collision risk: Hide ID could theoretically collide with symbol names.

No. A symbol cannot start with a digit. No fix needed.

13.18

Deep macro nesting obscures errors: Difficult debugging.

Yes. Debugging the C source can be done with gdb. Only the command bt is needed in order to see where the error occurred. No fix needed.

13.19

Generic error messages: Limited context for users.

Yes. If an error occurs in a Joy program, the possibility exists to run the program with trace enabled. There are two: `-d` and `-t`. They should give enough context. No fix needed.

13.20

No stack trace in error messages.

Yes. The stack traces of Java and C++ are infamous. When there really is an error, a stack trace is not possible. Same as in gdb: No stack. No fix needed.

13.21

No test coverage metrics: Cannot measure completeness.

Yes. The coverage was part of CI, but as stated before, CI is not what it used to be. The coverage has to be checked locally, after every change. No fix needed.

Here is the output of lcov, run locally:

```
Overall coverage rate:
  source files: 250
  lines.......: 100.0% (3326 of 3326 lines)
  functions...: 100.0% (342 of 342 functions)
Filter suppressions:
  missing:
    14 instances
Message summary:
  1 warning message:
    range: 1
```

13.22

Shell escape: security problem.

Yes. This is not available on WINDOWS_S, but WINDOWS_S only allows programs that can be downloaded from the Microsoft Store and Joy will not be placed there. So, shell escape can be added, but only if desired. Must fix.

14

The review also comes with a number of recommendations:

14.1

Add universal allocation error checking.

Yes. In general, error return values should be checked.

14.2

Implement recursion depth limit.

Yes. Whether the user is informed by the C programmer or by the OS makes no difference to the user. In both cases, either the user needs to split the problem or the Joy programmer needs to use an iterative algorithm. No action needed for the reference implementation, but of course other implementation can decide to add such a check.

14.3

Fix symbol.c: Add argument to `abortexecution_()`

No. This call is within a comment. It must be, because otherwise the source could not be compiled.

14.4

Add resource cleanup on error paths: Close files, free temporaries before longjmp.

No. There is no error in the reference implemention, until proven otherwise. No temporary files are used and the file that is closed before the longjmp should be closed after the longjmp.

14.5

Unify memory model code: Abstract NOBDW/BDW differences behind common interface.

No. Joy and joy1 are two separate repositories, although they have common code. There is no need to add code that distinguishes the

one from the other. It is sufficient to use conditional compilation, as provided by the C preprocessor.

14.6

Automate test execution: Integrate with CTest for CI.

Yes. Tests need not be integrated in CI. But of course, other implementations can do that. For the reference implementation, this would add too much complexity.

14.7

Add C-level unit tests: Test primitives directly.

Yes. Builtins are already tested in Joy. The advantage of that approach is that those tests can also be used in implementations that are built in other programming languages. The C code is adequately tested with static analyzers. Other implementations can add those unit tests, if desired.

14.8

Document thread-safety limitations: Clear warning in README.

No. The reference implementation has not been setup as a multi-threaded application. There is no need for a warning.

14.9

Refactor aggregate operations: Extract common map/filter/split logic.

No. There are already too many macro's and adding more just to show that some code is similar to other code is not needed.

14.10

Enhance error messages: Include source line numbers, stack depth.

No. Line numbers are already present in errors during scan time. Errors during runtime are unrelated to lines in the source code. Besides, the program can be run with trace enabled and then sufficient context is given around the error.

14.11

Add depth limits: Guard DISPLAYMAX, INPSTACKMAX with proper errors.

No. DISPLAYMAX and INPSTACKMAX are already guarded with a message. But of course, other implementations can be as verbose as they want.

14.12

Implement dynamic sets: Remove 64-element limitation.

Yes. The language already allows sets of different width. The function `setsize` can be used to see what the current implementation uses. Thus, this would be a good addition, just not for the reference implementation.

14.13

Add stack trace generation: Improve debugging experience.

No. The problem is, that when there is a real problem, the output of such a trace will be unreadable, that is: infinite. Also, there are already trace options `-d` and `-t`.

14.14

Profile and optimize: Identify hot paths, consider inlining.

Yes. Profiling can be done and this does not tell anything new. The reference implementation uses data structures that are flexible and such flexibility comes with a performance cost. The only way to avoid these costs is to remove all flexibility, as in the Legacy version, and be content with that. It does require extra failure messages and the use cases of the implementation are more limited.

14.15

Document memory layout: Add diagrams for GC phases.

Yes. The more documentation, the better. Not that anyone reads it.

14.16

Add configuration options: Expose limits as compile-time settings.

Yes. Traditionally, these options are in the file `globals.h`. They could be inserted in a separate config file, that is then included in `globals.h`.

Conclusion

None of the recommendations, except the first one, are needed to improve the Joy repository. Every addition to the code should be rejected, unless there is an error that must be corrected. More code does not improve the implementation. But of course, other implementations can add as much code as needed.

The aim of the reference repository is to strife for imperfection. There should be something left to be desired. A solution that is 99% correct is sufficient.

15

As mentioned, G3 currently uses 42minjoy. That Joy implementation is substandard. Notable differences with the full version of Joy are:

- no sets
- no floating points
- no CONTS
- no finclude
- different boolean test
- different identifiers
- unicode support
- other differences

In addition to CONTS, 42minjoy does not have any keywords. The only syntax that 42minjoy has are `==`, the semicolon and the full stop.

The other differences make it currently impossible to use the full version of Joy on the G3 sources. The other differences were resolved and G3 is now started from the full version.

4

Krybot

Joy

Some AI generated stuff. Mostly repetitive and sometimes useful.

Beginner Topics

These topics discuss the syntax of Joy and some of the builtins.

1. Introduction to Joy: What is Joy and Why Learn It?

The example program:

```
[1 2 3 4] [dup *] map sum
```

is not complete. Missing is a full stop at the end of the program. Without that stop, the language processor doesn't do anything. This needs to be mentioned in the first example. It is ok to omit the full stop in further examples.

2. Setting Up Your Joy Development Environment

Step 1: Downloading the Joy Interpreter

```
git clone https://github.com/Wodan58/Joy
cd Joy
make
```

The makefile has been setup with gcc; Mac users will probably want to use clang. That is a small change in the makefile.

Windows users can download the G3 installer from this location: G3-JOY-1(https://github.com/Wodan58/G3-JOY-1). It installs G3 and also joy.

Step 2: Running Joy

```
./joy
```

There is no prompt. Windows users who downloaded the installer can use the shortcut on the desktop to start Joy or use the shortcut in the list that is accessable from the start button or create a new shortcut.

3. Your First Joy Program: "Hello, World!"

```
"Hello, World!" print
```

No, Joy does not have print. Joy uses putchars. In general, names in Joy are as descriptive as possible. Also, this print is not a combinator. A combinator in Joy takes one or more quotations as parameter.

4. Understanding Joy's Syntax and Structure

```
42          // pushes the number 42
```

No, the character that is used to comment out the rest of the line is #, not //.

```
[null]      # base case: is the number zero?
[pop 1]     # base result: return 1.
[dup pred]  # recursive argument: n-1
[*]         # combine: multiply n with result
linrec
```

Yes, this computes the factorial of a number on the top of the stack.

5. Introduction to Joy's Functional Paradigm

```
DEFINE fact == [dup 1 >] [dup 1 - fact *] [] ifte.
```

This computes the factorial function all right, but the condition is not formulated as it should be: `[1 >]`. Joy sets up an hypothetical world where the condition is evaluated for the sole purpose of creating a boolean value. After reading that value, the hypothetical world is left to the garbage collector in order to be recycled.

6. Working with Numbers in Joy: Basic Operations

Modulo

```
10 3 mod
```

No, Joy does not have mod, it has rem. They are different, when operating on negative numbers:

```
-17 3 rem
```

Programming languages that have both rem and mod make the distiction that rem takes on the sign of the dividend, whereas mod takes the sign from the divisor. And rem uses truncating division towards 0, whereas mod uses floor division, towards -∞. It so happens that Joy prints -2 in the example, making the name rem appropriate.

7. Understanding Joy's Stack-Based Nature

```
+ : a b -> c    # consumes two elements and pushes one
dup : a -> a a  # duplicates the top of the stack
```

Yes, before and after pictures of the stack are useful.

8. Using the Basic Operators in Joy: +, -, *, /

/ is described as integer division. That is true when both parameters are integers. When either one of them is a floating point, the result will also be a floating point.

```
20 3 /      # result is 6
20.0 3 /    # result is printed as 6.6667
20 3.0 /    # likewise
20.0 3.0 /  # likewise
```

9. Variables and Constants in Joy: Assignment and Naming

```
DEFINE square == [dup *].
```

No, the syntax is: name == body, not: name == [body]. This definition of square is possible, not useful, as indeed, it requires i to evaluate.

Naming is limited to defining new words using DEFINE. No, apart from DEFINE, there is also HIDE and MODULE. They introduce local names.

10. Introduction to Functions in Joy: Defining and Calling Functions

Functions can be called by calling them by name:

```
5 square.
```

Square is not defined in one of the libraries, so it needs to be defined first:

```
DEFINE square == dup *.
5 square.
```

11. The Power of Composition in Joy: Combining Functions

No, Joy does not have compose and does not need it: concat will do. If the task is to:

- square a number
- add 1
- double the result

The following definitions are sufficient, because succ is builtin:

```
DEFINE square == dup *;
       double == 2 *.
```

Example:

```
7 square succ double.
```

And yes, DEFINE is followed by a list of definitions, separated by semicolon and terminated by a full stop.

```
[dup 1 <=]
[1]
```

```
[dup 1 -]
[*]
linrec
```

This computes the factorial of a number and has a different stack effect from the one presented in section 4. This one is nullary, whereas the one in section 4 is unary. A more idiomatic version is this:

```
[small]
[1]
[dup pred]
[*]
linrec
```

Linrec is the work horse combinator of Joy. More specialised combinators can also compute the factorial function and some of them are even more idiomatic than this one.

12. Understanding Joy's Recursion Model

```
DEFINE factorial ==
    [dup 1 >]
    [dup 1 - factorial *]
    [pop 1]
    ifte.
```

Yes, the factorial function keeps coming back. It is the "hello, world" of functional programming languages. This one is similar to the one in section 5, except the [pop 1] before the ifte. So, a 1 is popped and then pushed back. This is not needed, except that if the requirement is that als 0! must be computed correctly, then it is needed: $0! = 1$, by agreement.

```
DEFINE fib ==
    [dup 2 <=]
    [pop 1]
    [dup 1 -]
    [1 -]
    [+]
    binrec.
```

No, binrec takes 4 quotations as parameter, not 5. Also, when the third and fourth quotations are combined with a swap in between, it gives

wrong answers. A possible correction is:

```
DEFINE fib ==
    [2 <]
    [pop 1]
    [pred dup pred]
    [+]
    binrec.
```

The result of 10 fib is 89 and that is a Fibonacci number, although different from the fib in numlib.joy. The culprit is the second quotation. It should be empty. Also, the condition can be [small] instead of the one given.

```
DEFINE factorial ==
    [dup 1 <=]
    [pop 1]
    [[dup 1 -] factorial *]
    [null]
    genrec.
```

This program causes a Segmentation fault in the interpreter, something bad. The fourth quotation is suspicious: null is a predicate. The third is also suspicious: if genrec handles recursion, then why is there an explicit call to factorial? Indeed, the definition given in the Rationale document is:

```
[null] [succ] [dup pred] [i *] genrec
```

The Segmentation fault will be put on the TODO list. The interpreter should give an error message. It's a stack overflow and sometimes there is no message. Maybe a stack overflow during the print of a message?

```
DEFINE sum ==
    [null] [0] [uncons] [+] linrec.
```

No, this doesn't work. It gives a runtime error. The second quotation should be: [pop 0].

```
DEFINE map ==
    [null] [pop []] [uncons] [swap cons] linrec.
```

No, this doesn't work. It gives a runtime error. The fourth quotation should be [cons]. Also, the second quotation can be []. The first quotation sees an empty list and in response, the second quotation does not have

to do anything. But this map doesn't do anything. It takes a list apart and then builds it up again. Not useful. But there is a map builtin that does someting useful. It applies a function to each element of an aggregate and builds a new aggregate of the same type.

13. Working with Lists in Joy: Creating and Manipulating Lists

Reversing a list:

```
[] [1 2 3] [swap cons] fold
```

No, this doesn't work. The list is returned unmodified. Fold is not suitable for the job. It reduces an aggregate to a scalar. Reverse and reverselist are defined in seqlib.joy. A more elegant version:

```
DEFINE reverselist ==
    [] swap infra.

[1 2 3] reverselist.
```

Infra executes a quotation on a secondary stack, initially empty. When this stack is converted to a list, it appears inverted, as usual.

```
member
```

No, Joy doesn't have member.

14. Basic Control Flow in Joy: Conditional Execution

```
true [1] [2] if
```

No, Joy does not have an if combinator. The combinator that produces the desired result is branch. The word if only refers to the condition, whereas branch is better in describing the control flow.

```
empty
```

No, Joy does not test emptyness of a set with empty. Null is generic and can be used for sets.

```
[dup] [rollup >] [swap] [] ifte
```

No, this is not the min function. Joy already has min builtin, but if it needs to be defined, then this is better:

```
[<] [pop] [popd] ifte
```

Given two numbers, it will leave behind the smaller of the two.

```
-4 [dup 0 <] [-1] [dup 0 =] [0] [1] ifte ifte
```

The syntax is not correct. Ifte takes three quotations as parameter and the second ifte fails to do so. A better approach is to use the builtin compare:

```
-4 0 compare
```

There we go again, another factorial:

```
DEFINE factorial ==
    [dup 1 >]
    [dup 1 - factorial *]
    [1]
    ifte.
```

This one is nullary: it takes a number, calculates the factorial and pushes the result. The number is still there.

15. Looping in Joy: Repetition with Recursion

```
DEFINE countdown ==
    [dup 0 >]
    [dup . 1 -]
    [pop]
    linrec.
```

No, this doesn't work. The full stop ends a program or a definition. It is syntax. Also, linrec takes 4 quotations, not three. The desired effect can be reached with:

```
"numlib" libload.

DEFINE countdown ==
    [positive]
    [dup putln pred]
    while pop.
```

Yes, this is iterative, not recursive.

```
DEFINE facttail ==
    [dup 1 >]
    [[dup 1 - swap over * swap]]
```

```
    [pop]
    tailrec.
```

Oh no, the factorial again. This one doesn't work. The second quotation is returned as the result. It is certainly possible to use tailrec:

```
DEFINE facttail ==
    1 swap              # install accumulator
    [1 <=]              # test the number
    [pop]               # if smaller or equal to 1, done
    [dup pred [*] dip]  # multiply with the accumulator
    tailrec.
```

The function uses an accumulator, collecting the result. The number given as parameter is used in a countdown setting, until it reaches 1.

```
DEFINE factorial ==
    [1]                 # base case: 0! = 1
    [dup 1 - swap *]    # recursion: n * (n-1)!
    primrec.
```

No, this doesn't work. The swap before * is also suspicious. Multiplication is commutative and that means that swap is not necessary. The countdown is embedded in primrec. So, the correct definition is:

```
DEFINE factorial ==
    [1]
    [*]
    primrec.
```

Now, that is a beautiful definition! The library file numlib.joy contains yet another definition of factorial, using the times combinator.

```
DEFINE fib ==
    [dup 2 <]
    [1]
    [[dup 1 - fib] [swap 2 - fib] dip +]
    [i]
    genrec.
```

No, this doesn't work. Genrec already handles the recursion, making the direct calls to fib redundant. The correct definition is:

```
DEFINE fib ==
    [small]
```

```
[pop 1]
[pred dup pred]
[unary2 +]
genrec.
```

With the exception of unary2 in the fourth quotation, this solution is similar to the one that uses binrec. For a more wide spread definition: the second quotation should be empty. This generates the sequence: 0, 1, 1, 2, 3, 5 ... Starting with 1 pair of rabbits that become fertile and reproduce after two months, there will be two pairs after two months. Counting from 0 onwards - Fibonacci reintroduced 0 in Europe - the sequence gives the number of rabbits after n months.

16. Introduction to Joy's if and while Constructs

```
[0 >] [neg] [i] ifte
```

Joy has the abs function builtin. If it needs to be defined, then this is a more elegent definition:

```
[positive] [neg] [] ifte
```

If the number is 0 or negative, nothing needs to be done. That's why the third quotation is empty.

```
5 [dup 0 >] [dup . 1 -] while drop
```

No, this is a syntax error, as already mentioned. Joy has drop, but with a different meaning. What is needed here is pop.

17. Handling Errors and Exceptions in Joy

Nullary, unary, binary, ternary are combinators that guarantee that 0, 1, 2, 3 elements from the stack are destroyed. They are not predicates, testing whether the stack is sufficient large.

```
DEFINE safe-divide ==
    [swap 0 =] [drop "Error: Divide by zero"] [/] ifte.
```

No, this does not work. The condition should be: [null], testing the divisor, not the dividend. Also, drop cannot be used on an integer. This should be pop. Finally, pushing s string is not helpful. Output of the string with putchars is more appropriate.

```
[ [null] ["Empty"]
```

```
  [small?] ["Too small"]
  [large?] ["Too large"]
  [] ["OK"]
] cond
```

This will give a runtime error. Cond expects a list of lists. When correcting that, it needs a definition for small? This is surprising, as names are usually built from alphanumerics and the underscore, with an exception for the first char: it cannot be a digit but it can be a special character such as a question mark. However, it is accepted now by Joy. No reason why not.

```
DEFINE large? == 1114111 >;
       test   == [ [[null] "Empty"]
                   [[small] "Too small"]
                   [[large?] "Too large"]
                   ["OK"]
                 ] cond.
```

The number mentioned in large? is the largest Unicode code point.

18. Understanding the Joy Stack: Manipulating and Accessing Stack Elements

```
1 2 3 rot
```

Joy does not have rot. It has rotate and the result would be: 3 2 1. The middle element stays put and the outer elements change place. Joy also does not have roll, it has rotate.

```
1 2 3 rollup dup rolldown
```

The claim is made that a copy of the third element, 3, is placed on top. If 1 2 3 is evaluated, 3 will be on top and not be the third element. It doesn't work. If the third element is 3, then the following sequence does the job:

```
3 2 1 rotate dup rotated
```

The rotate brings the 3 on top, as required, duplicates it and brings the old stack in order by rotating again. AI cannot be asked to do this kind of shuffling. On the other hand, AI is good in explaining why it can't do it.

```
DEFINE dup2 == [dup swap dup swap].
```

No, that is not dup2, whatever the specifications are. The task is to duplicate the top two elements. If these elements are named A B, then the result can be A A B B, or it can be A B A B.

In inilib.joy it is defined as:

```
dup2 == dupd dup swapd;
```

This results in A B A B, because of the swapd. It could have been defined as:

```
dup2 == over over;
```

This is shorter. But then, over was not builtin when inilib.joy was created.

19. Printing Output in Joy: Displaying Values with . and print

Print is introduced as a way to print the top of the stack, without removing the value. Print is not a builtin of Joy, but could be defined as:

```
DEFINE print == dup put.
```

The full stop at the end of a program is syntax. It is there that the Joy program ends. It is from here that the REPL takes over and prints the top value of the stack, and also removes it, or it prints the entire stack without removing anything, or it does nothing. All of this based on the setting of autoput. The computation that Joy did may have been purely functional but as soon as the REPL takes over, it becomes imperative. Get and put are not purely functional. They were added to Joy for debugging purposes. This print as defined above is a debugging aid.

20. Using Joy's dup, drop, and swap Operations

It was already mentioned that Joy uses pop, not drop. Drop is something different. These operators are vary basic and it comes as a surprise that they are mentioned in section 20. They should have been treated in an earlier section, and maybe they were. Dup creates data, pop destroys it and swap changes the order of parameters: for subtraction and division this can be useful, if parameters have arrived in the wrong order.

Intermediate Topics

These topics discuss the more difficult algoritms and data structures.

21. Understanding Joy's Composition Operator: Building Complex Functions

```
[[dup +] [1 +] concat] is double-plus-one
5 double-plus-one i
```

No, Joy does not have runtime naming with is. Also, the i only executes the concat. It would take another i to also execute the concatenated quotation.

22. The Power of Functional Programming in Joy

```
5 [3 +] dip
```

This doesn't work. When the + is executed, the stack only contains 3 and a runtime error will follow.

23. Advanced List Operations in Joy: head, tail, and cons

Joy doesn't have the head operator. It is called first in Joy. Joy also does not have the tail operator. It is called rest in Joy.

```
[1 2 3 4] dup first swap rest [cons] swap dip
```

No, this doesn't work. The aim is to produce [2 3 4 1], moving the first value to the last position. AI cannot be asked to do this kind of shuffling, because it does not have access to a Joy binary that it can use to do the math.

```
[1 2 3 4] unswons   # this isolates the head, head on top
unitlist concat     # this appends the head to the tail
```

Unitlist is defined in agglib.joy and that library is loaded at startup.

24. Recursion vs Iteration in Joy: Choosing the Right Approach

Nothing new here.

25. Working with Multiple Arguments in Joy Functions

```
DEFINE twice == [dup i i].

3 [succ] twice.
```

This should produce 5. It doesn't work. Joy has a program called j-nestrec.joy that has twice defined as:

```
DEFINE twice-i == dup [i] dip i.

3 [succ] twice-i.
```

26. Understanding Joy's map and reduce Operators

```
[1 2 3 4] [+] reduce
[2 3 4] [*] reduce
```

Joy does not have reduce and such a reduce is not possible. That is because the base case differs, depending on the operator. In case of plus it is 0 and in case of * it is 1. So, the base case must be mentioned explictly and that is why the definitions of sum and product differ:

```
sum == 0 [+] fold
product == 1 [*] fold
```

Sum is defined in agglib.joy, loaded at startup; product is defined in seqlib.joy that must be loaded explicitly:

```
"seqlib" libload.
```

So, a reduce combinator cannot exist in Joy.

27. Creating and Using Custom Operators in Joy

Nothing new here.

28. The Role of the Stack in Joy: Deep Dive

Nothing new here.

29. Understanding Joy's join and split Operations for List Manipulation

Joy does not have a join combinator. It has flatten defined in seqlib.joy

```
[[1 2] [3 4] [5]] flatten
```

Joy has split with a different meaning:

```
[1 2 3 4 5] 3 split
```

So, how can this be defined in Joy, and how should it be named? The operation is to take 3 elements, drop 3 and pair the result:

```
DEFINE take-drop ==
    dup2 drop [take] dip pairlist.
```

Another example:

```
[[[a] [b]] [[c]] [d]]]
```

This example can be flattened to [a b c d] by applying flatten twice or by executing treeflatten. Both are defined in seqlib.joy

30. Introduction to Joy's quote and eval for Meta-Programming

Quoting in Joy uses the [] mixfix operator. It is neither prefix, nor infix, nor postfix. Eval in Joy is just i. A LISP system has eval and apply. Joy has only eval. And i is not the only combinator that can evaluate quotations. It happens to be the simplest.

```
[dup cons] dup cons
```

This is not a real Quine. A Quine is a program that prints its own source code. It can be done in all programming languages. A real Quine in Joy:

```
"dup put 32 putch putchars 10 putch."
 dup put 32 putch putchars 10 putch.
```

This works, when stored in one line, because of two ways to output a string: putchars prints the content, whereas put outputs the string with surrounding quotes.

31. Using Joy's rotate, unrot, and reverse for Stack Manipulation

```
1 2 3 rotate .s     # 2 3 1
```

Joy has rotate, not this one. And the .s is also not part of Joy. And Joy also does not have unrot.

32. Using Joy's first and second for Tuple Operations

```
DEFINE map2 ==
    [uncons] dip
    swap
    [swap] cons cons.
[2 3] [1 +] map2.   # [3 4]
```

This doesn't work. The function [1 +] is not executed. The normal map works just fine:

```
[2 3] [succ] map.
```

The tuple-swap also does not work.

```
[7 9] tuple-swap.   # [9 7]
```

Here is a definition that does work:

```
DEFINE tuple-swap ==
    i               # if the tuple consists of integers
    swap
    pairlist.
```

The distance function also does not work:

```
DEFINE distance ==
    [uncons swap uncons]    # x1 y1 x2 y2
    [rot - dup *] dip       # (x2 - x1)^2.
    - dup * + sqrt.
```

This one does:

```
DEFINE distance ==
    i swap unswonsd uncons  # [y2] x2 x1 [y1]
    [- dup * swap] dip      # (x2 - x1)^2 [y2] [y1]
    [first] dip first       #      ,,     y2 y1
    - dup * + sqrt.

[1 2] [4 6] pairlist distance.
```

33. Pattern Matching in Joy: Simplifying Function Calls

Joy has a builtin case, similar in functionality to cond.

```
[[1 2] [3 4] [5 6]] [[+]] map.
```

This doesn't work as expected. But this one does:

```
[[1 2] [3 4] [5 6]] [i +] map.
```

The i unpacks the quotation and that gives + two parameters to build on. The map builds a new list of three items with whatever it finds on top.

34. Optimizing Recursive Functions in Joy

Joy does not have nip. It has popd. Apart from that: Nothing new here.

35. Building Higher-Order Functions in Joy

Joy has filter as builtin. Apart from that: Nothing new here.

36. Function Composition and Currying in Joy

```
3 [dup *] [succ] cleave.
```

This one actually works. Apart from that: Nothing new here.

37. Using Joy's not, and, and or for Logical Operations

```
[even] [positive] and
```

No, that doesn't work. And takes booleans or sets as parameters, not lists.

38. Understanding the while and until Constructs in Joy

Joy does not have until.

39. Building List Transformers with Joy: Mapping, Filtering, and Reducing

```
[[1 2] [3 4] [5 6]] [[+] fold] map.
```

No, that doesn't work. Fold needs three parameters. This one does:

```
[[1 2] [3 4] [5 6]] [sum] map.
```

Sum is defined as: 0 [+] fold in agglib.joy.

40. Introduction to Joy's factorial and fibonacci Examples

Joy does not have rec.

41. Managing State in Joy: The Use of Functions with Side Effects

Joy does not have set.

42. Introduction to Joy's collect for Grouping Results

Joy does not have collect.

43. Working with Nested Functions and Closures in Joy

Nothing new here.

44. Using Joy's replicate for Function Application

Joy does not have replicate.

```
5 42 replicate
```

This should produce a list with 5 times 42. Joy does have times. It expects a quotation that is executed a number of times. Here it is:

```
DEFINE replicate ==
    [] rollup [swons] cons times.
```

It works for the example. It may not work for all purposes that replicate should have.

45. Creating Custom Data Structures in Joy

Joy has native tree support. The quotations that Joy uses are hierarchical in nature. And Joy has three combinators, specifically designed for trees:

```
treestep, treerec, and treegenrec
```

46. Understanding Higher-Order Functions in Joy

Nothing new here.

47. Performance Optimization in Joy: Tail Recursion and Memoization

Nothing new here.

48. Creating Function Pipelines in Joy

Nothing new here.

49. Building and Using Modular Code in Joy

Nothing new here.

50. Exploring Joy's Memory Model and Garbage Collection

Nothing new here.

Advanced Topics

Most of these topics are geared towards making Joy useful. Does Joy really need to be useful?

51. Advanced Functional Programming Techniques in Joy

Nothing new here.

52. Using Advanced Recursion Patterns in Joy

There might be something here.

53. The Joy Type System: Understanding Types and Type Inference

Nothing new here.

54. Working with Streams and Infinite Sequences in Joy

Joy has support for this in lazlib.joy

55. Implementing Complex Algorithms in Joy

Nothing new here.

56. Building a Simple Parser with Joy

Joy has support for this in grmlib.joy

57. Understanding Joy's quote and eval for Code Execution

Nothing new here.

58. Managing Scope in Joy: Local vs Global Variables

Nothing new here.

59. The Role of Currying and Partial Application in Joy

Nothing new here.

60. Functional Design Patterns in Joy

Nothing new here.

61. Advanced Stack Manipulation Techniques in Joy

Joy does not have dupdip or conswap. Joy does not have spread. It also does not have enstack and destack. It does have stack and unstack.

62. Working with Data Flow Programming in Joy

Nothing new here.

63. Interfacing Joy with External APIs and Libraries

Joy does not have an FFI. The only way that Joy can be extended is by adding a new builtin that wraps the behaviour of an external function and makes it accessible to Joy.

64. Parallelism and Concurrency in Joy

Joy does not have fork, join, channel as builtin. Experiments have shown that it does not improve performance. It adds complication. It does not add to what can be computed with Joy. Multi-tasking only complicates and is better left to the operating system. A programming language that is designed to allow to build applications does not need multi-tasking, unless it is a browser.

65. Building Interactive Applications with Joy

Joy has fgetch, not getc. Also, it has putchars, not puts. Joy has a library that contains functions for user interaction: tutlib.joy

66. Using Joy for Symbolic Computation

Joy has a library for this: symlib.joy

67. Building and Using Domain-Specific Languages (DSLs) in Joy

Nothing new here.

68. Implementing a Simple Language Interpreter with Joy

Joy does not have an example of an imperative language or calculator. It does have a lisp interpreter in lsplib.joy with the usual eval and apply and it has a joy inner interpreter in jp-joyjoy.joy.

69. Advanced Error Handling and Exception Management in Joy

Nothing new here.

70. Creating Efficient Sorting and Searching Algorithms in Joy

Insertion sort is explored in this blogpost: Verzettelung. The coding that AI does cannot be trusted. The link made visible:

```
https://verzettelung.com/22/07/20/
```

71. Working with Large Data Sets in Joy

Nothing new here.

72. Building Data Pipelines with Joy

```
[1 2 3] [4 5 6] [[+]] zipwith
```

No, this doesn't work. Joy does not have zipwith, it has mapr2 in agglib.joy:

```
[1 2 3] [4 5 6] [+] mapr2
```

The result:

```
[5 7 9]
```

73. Using Joy for Web Development: Connecting to Servers and Databases

Nothing new here.

74. Integrating Joy with JavaScript for Web Applications

Nothing new here.

75. Implementing Advanced Algorithms in Joy: Graphs, Trees, etc.

Nothing new here.

76. Introduction to Metaprogramming in Joy

Nothing new here.

77. Using Joy for Machine Learning: Building Models with Functional Programming

Nothing new here.

78. Exploring Joy's Limitations and Performance Bottlenecks

Area	Limitation
Variables & Scope	No local/global vars, no closures
Stack Manipulation	Verbose and hard to read
Execution Model	Interpreted, no optimization
Standard Library	Minimal and outdated
Concurrency	No threads or async
Error Handling	No try/catch or safe failure paths
System Integration	No FFI or external bindings
Tooling	Minimal IDE/editor support
Type Safety	No types or contracts
Recursion	Stack based, no TCO

Interesting.

79. Functional Reactive Programming with Joy

Nothing new here.

80. Writing Tests and Test Suites for Joy Code

Nothing new here.

81. Building a Joy-based Web Scraper

Nothing new here.

82. Interfacing Joy with C and Other Languages

Nothing new here.

83. Writing Joy for Embedded Systems

Nothing new here.

84. Implementing Advanced Data Structures in Joy: Trees, Heaps, etc.

Joy has advanced data structures in typlib.joy. This file contains a stack, a bigset, and a dictionary. That should be sufficient to implement Joy in Joy.

85. Working with Randomness in Joy: Generating Random Data

Nothing new here.

86. Extending Joy: Writing Plugins and Extensions

Nothing new here.

87. Building Robust Applications in Joy: Error Handling Best Practices

Nothing new here.

88. Implementing a Simple Game Engine in Joy

Nothing new here.

89. Joy in Data Science: Performing Statistical Computations

Nothing new here.

90. Using Joy for Data Mining and Analysis

Nothing new here.

91. Integrating Joy with SQL and NoSQL Databases

Nothing new here.

92. Building Real-Time Applications with Joy

Nothing new here.

93. Using Joy for System Programming and Scripting

Nothing new here.

94. Functional Design and Architecture with Joy

Nothing new here.

95. Advanced Memory Management in Joy

Nothing new here.

96. Using Joy for Data Visualization: Graphs and Charts

Nothing new here.

97. Writing Distributed Systems in Joy

Nothing new here.

98. Building a Joy-based Web Server

Nothing new here.

99. Performance Tuning and Profiling in Joy

Nothing new here.

100. The Future of Joy: Trends, Community, and Ecosystem

Nothing new here.

Conclusion

Something can be learned from every program, even if it is buggy. That is why reading these chapters was useful.

Ruurd Wiersma

7th edition

26-01-26