Development

Release 0.0

Wolfgang Scherer

Mar 19, 2023

Quickstart

Abstract

CONTENTS

List of Figures iii

List of Tables iv

List of Code Blocks v

1 Introduction 1

2 Software development model 2

3 Control Structures 3

4 Logic 5
- 4.1 Logic symbols . . . 5
- 4.2 Notation . . . 5
- 4.3 Combinatorics . . . 5
- 4.4 Boolean functions and truth tables . . . 8
- 4.5 Mask operation . . . 10
- 4.6 Functional completeness . . . 11

5 Source Control Management 16
- 5.1 Kallithea . . . 16
- 5.2 Mercurial Repository Trimming . . . 16

6 Shell basics 17
- 6.1 Job Control . . . 17
- 6.2 POSIX . . . 17
- 6.3 Special Purpose Language vs. Generic Programming Language . . . 18
- 6.4 WRF loop - single line processing in shell . . . 20
- 6.5 Single quoting . . . 26
- 6.6 Construct correctly quoted shell script . . . 28
- 6.7 Command execution . . . 31
- 6.8 . command . . . 32
- 6.9 Subshell and compound commands . . . 32

7 Shell Wildcards and Filenames 35
- 7.1 Glob pattern expansion . . . 35
- 7.2 Non-matching glob patterns . . . 35
- 7.3 Directory separator and illegal characters . . . 36

8 Shell Tools 39
- 8.1 dir_to_dot.sh - generate dot(1) graph from directory . . . 39

9 Python 43
- 9.1 Language extensions . . . 43

10 Common Lisp 44

11 Evaluating algorithms 46
11.1 Big O notation 46
11.2 Bakado - the way of the idiot 47
11.3 Taidanado - the way of the lazy 48
11.4 Satori o aita tensaido - the way of the enlightened genius 48

12 Exchange variables without temporary variable 49
12.1 Bewerbungsgespräch 49
12.2 Hardware 49
12.3 Software 50

13 MOV is Turing-complete 57

14 Markup converter design 59
14.1 Direct conversion between markup languages 59
14.2 Conversion via central abstract DOM 60

15 shell PIDs 61

16 Emacs Extension 65
16.1 Emacs-Erweiterungen mit Cut-Paste-Modify 65

17 klicki-bunti-und-das-fliewatüüt 71

18 Shell Option Evaluation Loop 72
18.1 Shell Commmand Grammar 72
18.2 Option Loop Activity Diagram 73
18.3 Option Loop Template 73
18.4 Snippet for defining an option 74

Abbreviations 81

Glossary 82

Index 83

LIST OF FIGURES

3.1 activitiy diagram for file existence check 4

6.1 WRF loop 20
6.2 WRF loop with standard IFS split 22
6.3 WRF loop with special IFS split 23
6.4 Shell command execution process 32

8.1 dir_to_dot.sh generator output 40
8.2 dir_to_dot.sh generator script 41
8.3 list_dir recurvise graph generator 42

16.1 Hilfe zur Funktion `shell` 65

LIST OF TABLES

LIST OF CODE BLOCKS

3.1 duplicated init/cont code . . . 3
3.2 single instance of init/cont code . . . 3
3.3 file existence check . . . 3
6.1 WRF loop . . . 20
6.2 WRF loop with standard IFS split . . . 21
6.3 WRF loop with special IFS split . . . 23
6.4 AWK script to split and process lines via callback . . . 24
6.5 Function split and process lines via callback . . . 24
6.6 Example for split and process lines via callback . . . 24
6.7 Script generated by example for split and process lines via callback . . . 25
6.8 Output from example for split and process lines via callback . . . 26

CHAPTER ONE

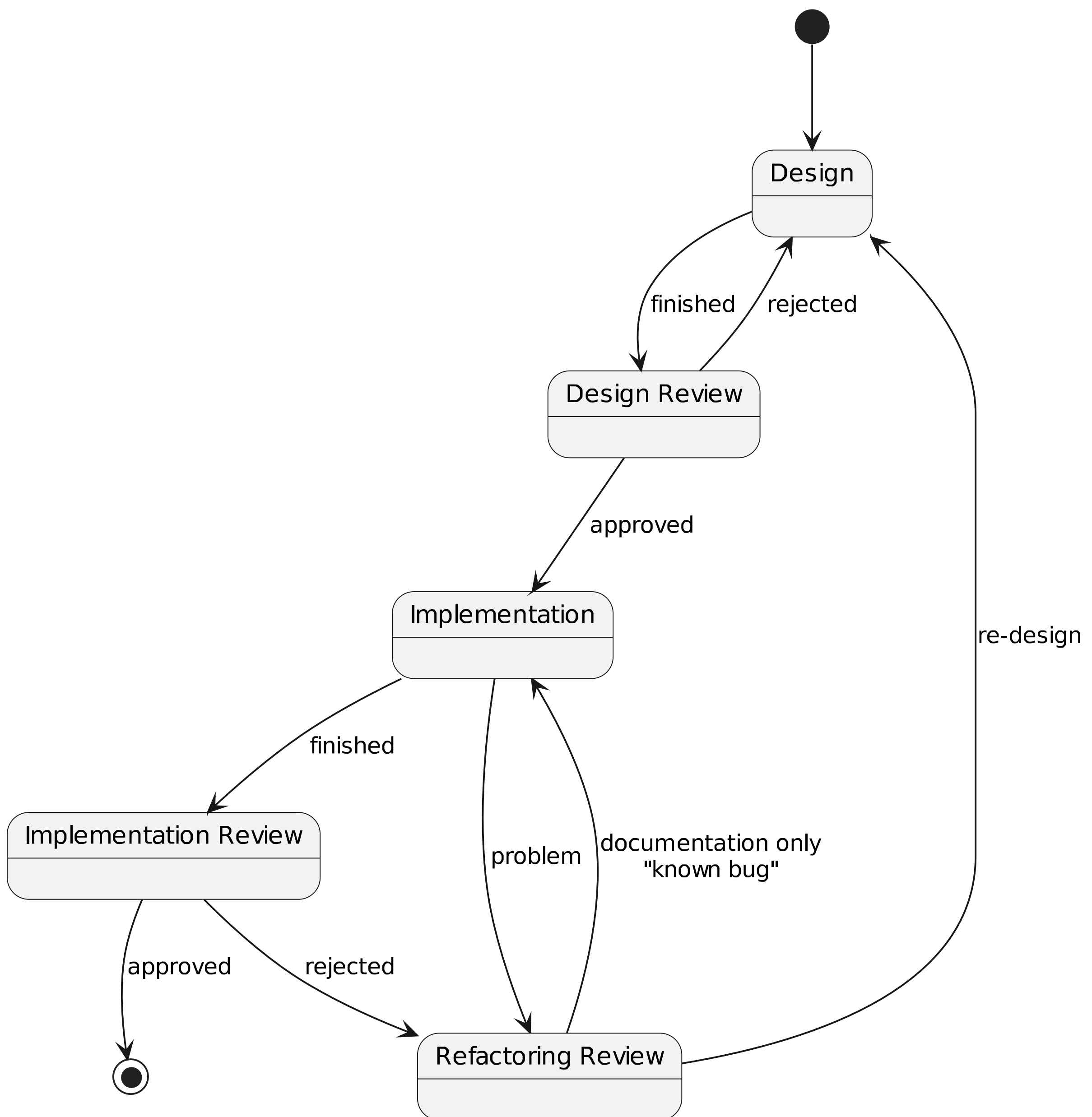
INTRODUCTION

Software development is as old as human history, it has given us many a religious belief system, used to control human brains, ultimately eliciting desired behavior in the attached human body.

CHAPTER

TWO

SOFTWARE DEVELOPMENT MODEL



CHAPTER THREE

CONTROL STRUCTURES

All control structures can be reduced to a conditional jump (see Control Structures (German)).

The unconditional while loop (endless loop) with a break before the repetition can be used to simulate complex free form case / when structures that do not fit the if / elseif cascade nicely.

With a conventional while loop, initialization/continuation code:is somethimes duplicated (see listing 3.1).

listing 3.1: duplicated init/cont code

```
1  ch = get_char()
2  while ch != EOF:
3      print(ch)
4      ch = get_char()
```

listing 3.2 shows, how this duplication can be avoided.

listing 3.2: single instance of init/cont code

```
1  while True:
2      ch = get_char()
3      if ch == EOF:
4         break
5      print(ch)
```

Another common problem is found in shell scripts, when it is necessary to determine whether one or more files actually exist (see listing 3.3).

listing 3.3: file existence check

```
1  # clear first_readable                                                   #a0 :;
2  first_readable=
3  # while (for each _file in screenlog.?) is (do)                           #a0
4  for _file in screenlog.?;
5  do
6      # if (_file is **not** readable?) then (yes)                          #a0
7      test -r "${_file}" ||
8          break                                                             #a0
9      # endif                                                               #a0
10     # set first_readable = "${_file}"                                     #a0 :;;
11     first_readable="${_file}"
12     break                                                                 #a0
13 done                                                                      #a0
14 # if (first_readable is empty?) then (yes)                                #a0
15 test -z "${first_readable}" && ...
16 # endif                                                                   #a0
```

figure 3.1 shows the logic behind the file existence check.

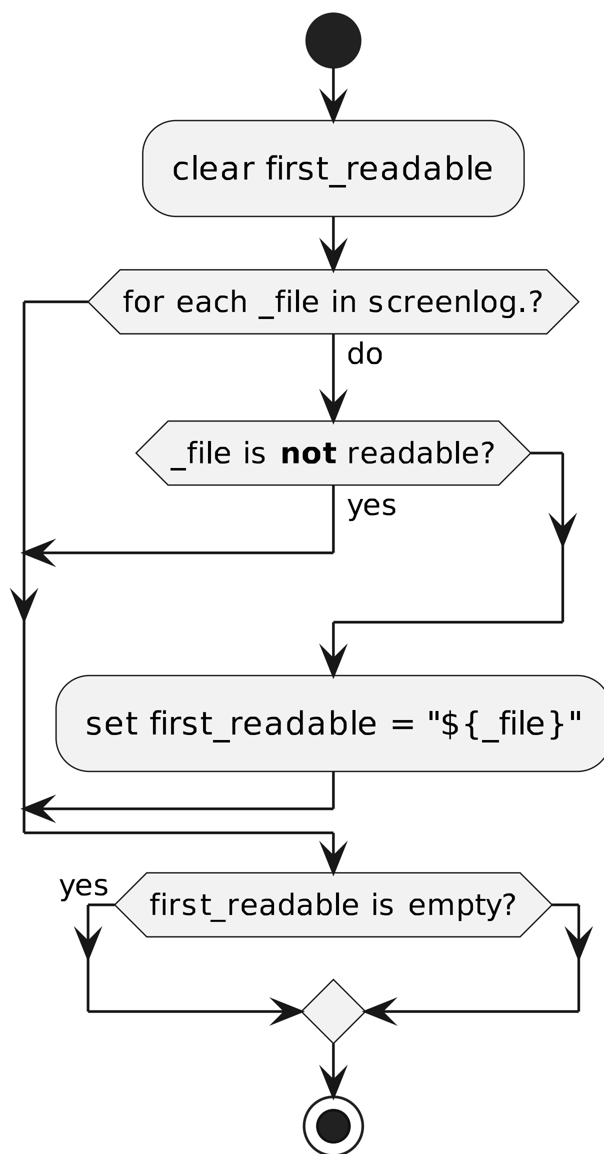


figure 3.1: activitiy diagram for file existence check

CHAPTER FOUR

LOGIC

On Wikipedia, the English entry Truth function has the most elaborate table of binary truth functions with various representations for all 16 functions. The German entry Logische Verknüpfung shows a systematic and complete table of all binary Boolean functions, while the English entry Logical connective does not.

The German entry Boolesche Funktion dicusses general arity of n-ary logical funtions for n=0 (truth values), n=1, n=2.

Wikiversity offers an interesting article about 3-ary Boolean functions.

4.1 Logic symbols

Some logical symbols used in this documentation. See also List of logic symbols on Wikipedia.

Symbol	Text	Name	Alternatives
$\neg$	`!`	NOT	$\sim$
$\wedge$	`&`	AND	
$\vee$	`|`	OR	
$\rightarrow$	`->`	IF	
$\leftrightarrow$	`<->`	IFF	$\Leftrightarrow$, $\equiv$
$\veebar$	`^`	XOR	$\oplus$, $\not\leftrightarrow$, $\not\equiv$
$\uparrow$	`!&`	NAND	
$\downarrow$	`!|`	NOR	
$\perp$	`F`	FALSE	`0`
$\top$	`T`	TRUE	`1`

4.2 Notation

For index notation of sequences, e.g.:

$$\underset{k=0}{\overset{n-1}{(x_k)}} = (x_k)_{k=0}^{n-1} = (x_0, x_1, \ldots, x_{n-1})$$

see article Indexing. For mathematical families in general see Indexed family or Familie (German).

For functions, see Notation for functions and Table of symbols for functions (German).

4.3 Combinatorics

Fundamental logic is mainly governed by variations without repetition (or n-sequences in X[1]):

[1] The German Wikipedia article Abzählende Kombinatorik has a table for permutations, variations and combinations with and without repetition. The English Wikipedia disambiguates variations as:

When the ordering of objects matters, and an object can be chosen more than once, we are talking about **variations with repetition**, and the number of variations is:

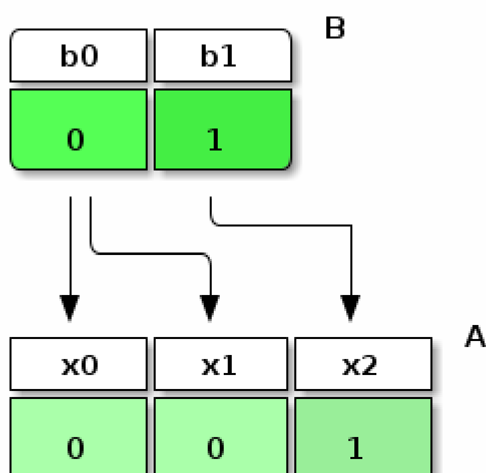
$$c^n$$

where c is the number of objects from which you can choose and n is the number of objects we can choose (repetitions allowed). (Source: Variations with repetition)

Distributing $c = 2$ balls from a sequence $B = (b_0, b_1) = (0, 1)$, to n boxes x_k, results in a distribution

$$A = \overset{n-1}{\underset{k=0}{(x_k)}}\,.$$

Since order matters and there can be more boxes than unique balls are available, the distribution must be **variations with repetition**. The following figure shows an example for distributing $c = 2$ balls to $n = 3$ boxes:

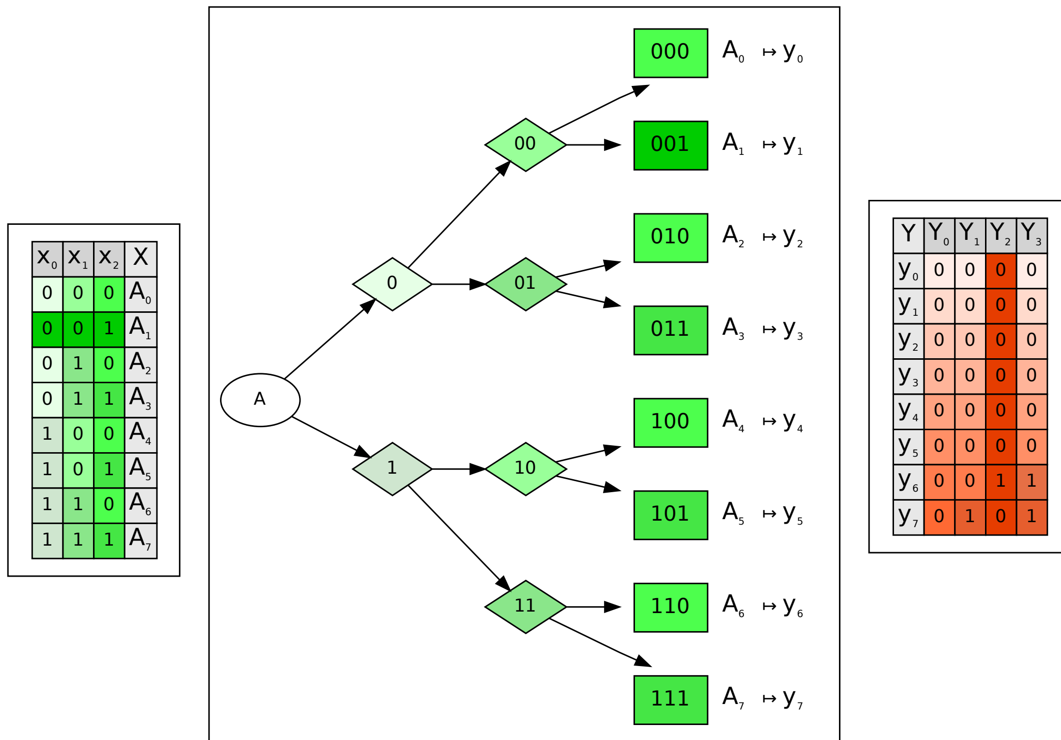


For $c = |B| = 2$ balls and $n = |A|$ boxes, the total number of possible variations A_j is c^n. The sequence X of variations A_j is then

$$X = \overset{2^n-1}{\underset{j=0}{(A_j)}}\,.$$

The following figure shows an example for $n = 3 \Rightarrow |X| = 8$:

Variations with repetition, a term in combinatorics commonly used by non-English authors for n-tuples which does not lead to the right place. The article Twelvefold way describes several classifications of combinatorial objects, with n-sequences in X as equivalent of variations with repetition.



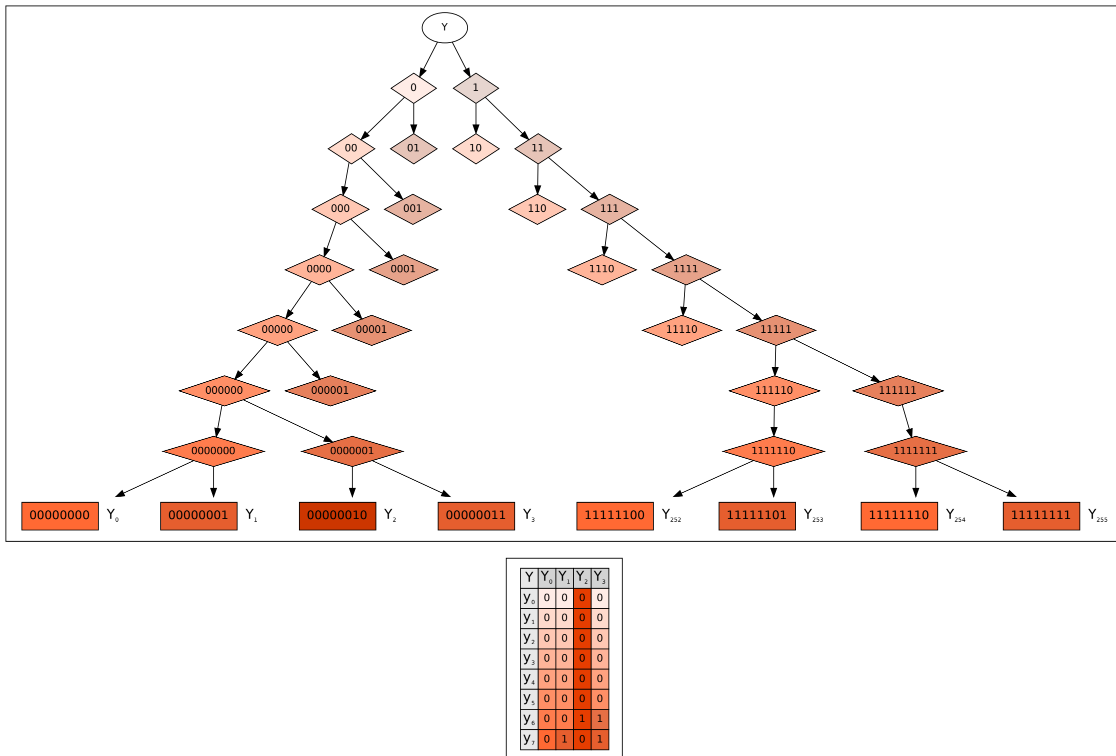
Correlating each distribution A_j to a new box y_j generates a sequence of boxes

$$Y = \left(y_j\right)_{j=0}^{2^n-1},$$

The number of possible distributions for 2 balls to boxes y_j is $2^{|Y|} = 2^{2^n}$, which generates a sequence of distributions

$$F = \left(Y_i\right)_{i=0}^{2^{2^n}-1}.$$

The following figure shows an example for $n = 3 \Rightarrow |A| = 8 \Rightarrow |F| = 256$:



Y	Y_0	Y_1	Y_2	Y_3
y_0	0	0	0	0
y_1	0	0	0	0
y_2	0	0	0	0
y_3	0	0	0	0
y_4	0	0	0	0
y_5	0	0	0	0
y_6	0	0	1	1
y_7	0	1	0	1

4.4 Boolean functions and truth tables

Quoting from article Boolean function:

> In mathematics and logic, a (finitary) Boolean function (or switching function) is a function of the form $: B^n \to B$, where $B = (0, 1)$ is a Boolean domain and n is a non-negative integer called the arity of the function. In the case where $n = 0$, the "function" is essentially a constant element of B.
>
> Every n-ary Boolean function can be expressed as a propositional formula in n variables $x_0, \ldots, x_n$.

Boolean functions are classified by arity $n \in \mathbb{N}_0$ as $f^n : B^n \to B^1$, such that each function f^n maps a sequence $A = (x_k)_{k=0}^{n-1}$ of n values $x_k \in B$ to a single value $y \in B$:

$$f^n : A \mapsto y.$$

The number of possible variations for A is $|(B \to A)| = |B|^{|A|} = 2^n$, which generates the sequence of possible input variations

$$X = \underset{j=0}{\overset{2^n-1}{(A_j)}},$$

and the corresponding sequence of output values

$$Y = \underset{j=0}{\overset{2^n-1}{(y_j)}}.$$

The number of possible output sequences is $|(B \to Y)| = |B|^{|Y|} = 2^{2^n}$. Therefore the sequence F^n of n-ary Boolean functions is

$$F^n = \left(f_i^n \middle| \begin{array}{l} f_i^n : X \to Y_i, \\ \quad A_j \mapsto y_j \end{array} \right)_{i=0}^{2^{2^n}-1}.$$

A truth table defines a function f_i^n by specifying an output value y_j for each combination A_j of input values x_k.

4.4.1 Universal lookup function

The order of sequences F^n, X, Y is defined such that each truth table row j

$$f_i^n(A_j) = y_j$$

can be calculated independently given arity n, function index i and output value index j.

Function $\text{bin}(d, m)$ converts an integer d into a sequence of binary digits:

$$\text{bin}(d, m) : d \mapsto (b_e | b_e = \left\lfloor \frac{d}{2^{m-1-e}} \right\rfloor \mod 2)_{e=0}^{m-1}, d \in \mathbb{N}_0, b_e \in B.$$

The sequence of output values Y_i for f_i^n is generated by

$$Y_i = \text{bin}(i, 2^n).$$

The truth table row j is then defined by

$$f_i^n(A_j) = Y_{i_j}.$$

Given f_{23}^3, the sequence of ouptut values Y_{23} is $(0, 0, 0, 1, 0, 1, 1, 1)$ and truth table row $j = 3$ is then defined by $f_{23}^3(0, 1, 1) = 1$. The follwing table shows the complete complete truth table for f_{23}^3:

$k \rightarrow$	0	1	2		
$j \downarrow$	x_0	x_1	x_2		f_{23}^3
0	0	0	0		0
1	0	0	1		0
2	0	1	0		0
3	0	1	1		1
4	1	0	0		0
5	1	0	1		1
6	1	1	0		1
7	1	1	1		1

4.4.2 Nullary Boolean functions

These are the truth constants from B expressed as functions $(f_i^0())_{i=0}^1$ without parameters.

f_0^0	f_1^0
$\bot$	$\top$
0	1

4.4.3 Unary Boolean functions

		f_0^1	f_1^1	f_2^1	f_3^1
p		$\bot$	p	$\neg p$	$\top$
0		0	0	1	1
1		0	1	0	1

4.4.4 Binary Boolean functions

			f_0^2	f_1^2	f_2^2	f_3^2	f_4^2	f_5^2	f_6^2	f_7^2	f_8^2	f_9^2	f_{10}^2	f_{11}^2	f_{12}^2	f_{13}^2	f_{14}^2	f_{15}^2
p	q		$\bot$	$\wedge$	$\nrightarrow$	p	$\nleftarrow$	q	$\veebar$	$\vee$	$\downarrow$	$\leftrightarrow$	$\neg q$	$\leftarrow$	$\neg p$	$\rightarrow$	$\uparrow$	$\top$
0	0		0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1		0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0		0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1		0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

4.5 Mask operation

Let $m, d, b \in B$, let $g_c^1 \in F^1$ be an arbitrary unary logical function. Then an operation $\mathrm{O}_{\text{mask}} : B^2 \rightarrow B^1$ is a mask operation, iff for all combinations of input bits $((m,d)_j)_{j=0}^{2^2}$ there exists a constant b so that function $f_i^2(m,d) = d$, if $m = b$, and $f_i^2(m,d) = g_c^1(d)$, if $m = \neg b$.

$$\mathrm{O}_{\text{mask}} : B^2 \rightarrow B^1 \Leftrightarrow \forall m, d \, \exists b : f_i^2(m,d) = \begin{cases} d & \text{if } m = b \\ g_c^1(d) & \text{if } m = \neg b \end{cases}$$

Truth table, using p as mask bit, q as data bit:

			f_0^2	f_1^2	f_2^2	f_3^2	f_4^2	f_5^2	f_6^2	f_7^2	f_8^2	f_9^2	f_{10}^2	f_{11}^2	f_{12}^2	f_{13}^2	f_{14}^2	f_{15}^2
m	d		$\bot$	$\wedge$	$\nrightarrow$	m	$\nleftarrow$	d	$\veebar$	$\vee$	$\downarrow$	$\leftrightarrow$	$\neg d$	$\leftarrow$	$\neg m$	$\rightarrow$	$\uparrow$	$\top$
0	0		0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1		0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0		0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1		0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Only those functions qualify as candidates for mask operations, which either have 2 entries for $f_i^2(m,d)$ with $f_i^2(0,0) = 0$ and $f_i^2(0,1) = 1$, or 2 entries with $f_i^2(1,0) = 0$ and $f_i^2(1,1) = 1$:

			f_1^2	f_4^2	f_5^2	f_6^2	f_7^2	f_9^2	f_{13}^2
m	d		$\wedge$	$\nleftarrow$	d	$\veebar$	$\vee$	$\leftrightarrow$	$\rightarrow$
0	0		0	0	0	0	0	1	1
0	1		0	1	1	1	1	0	1
1	0		0	0	0	1	1	0	0
1	1		1	0	1	0	1	1	1

4.5.1 Clearing data bits

f_1^2 (AND) allows clearing data bits by setting the corresponding mask value to 0. For mask values of 1, the data bits remain unmodified ($b = 1, g_c^1 = f_0^1$):

MASK	1	1	0	0	1	1	0	0
DATA	0	1	1	1	0	1	0	1
AND	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
RES	0	1	0	0	0	1	0	0

4.5.2 Setting data bits

f_7^2 (OR) allows setting data bits by setting the corresponding mask value to 1. For mask values of 0, the data bits remain unmodified ($b = 0, g_c^1 = f_3^1$):

MASK	0	0	1	1	0	0	1	1
DATA	0	1	1	1	0	1	0	1
OR	↓	↓	↓	↓	↓	↓	↓	↓
RES	0	1	1	1	0	1	1	1

4.5.3 Inverting data bits

f_6^2 (XOR) allows inverting data bits by setting the corresponding mask value to 1. For mask values of 0, the data bits remain unmodified ($b = 0, g_c^1 = f_2^1$):

MASK	0	0	1	1	0	0	1	1
DATA	0	1	1	1	0	1	0	1
XOR	↓	↓	↓	↓	↓	↓	↓	↓
RES	0	1	0	0	0	1	1	0

Special mask operations are:

$$\begin{array}{rcll} f_6^2(m, \ \ m) & = & (m \veebar \ \ m) & = 0 \\ f_6^2(m, \neg m) & = & (m \veebar \neg m) & = 1 \\ \neg f_6^2(m, \ \ m) & = & \neg(m \veebar \ \ m) & = 1 \end{array}$$

4.5.4 Other mask operations

The other mask operations can all be expressed in terms of AND, OR, XOR.

f_4^2 allows clearing data bits by setting the corresponding mask value to 1. For mask values of 0, the data bits remain unmodified ($b = 0, g_c^1 = f_0^1$). This can simply be replaced by inverting the mask bits and performing an AND operation:

$$f_4^2(m, d) = f_1^2(\neg m, d) = \neg m \wedge d$$

f_5^2 leaves all data bits unmodified ($b = 0, g_c^1 = f_1^1$ and $b = 1, g_c^1 = f_1^1$). It is therefore really a NOOP, i.e., just using the data bits is sufficient. However, to fulfill the requirements of a binary operation with arbitrary mask bits, any mask operation can be used by transforming the mask so as to leave the data bits unmodified.

$$\begin{array}{llll} f_5^2(m, d) & = f_1^2(f_6^2(m, \neg m), d) & = (m \veebar \neg m) \wedge d & = 1 \wedge d \\ f_5^2(m, d) & = f_7^2(f_6^2(m, \ \ m), d) & = (m \veebar \ \ m) \vee d & = 0 \vee d \\ f_5^2(m, d) & = f_6^2(f_6^2(m, \ \ m), d) & = (m \veebar \ \ m) \veebar d & = 0 \veebar d \end{array}$$

f_9^2 (IFF) allows inverting data bits by setting the corresponding mask value to 0. For mask values of 1, the data bits remain unmodified ($b = 1, g_c^1 = f_2^1$). This can be achieved, by inverting the mask for f_7^2:

$$f_9^2(m, d) = f_7^2(\neg m, d) = \neg m \veebar d$$

f_{13}^2 allows setting data bits by setting the corresponding mask value to 0. For mask values of 1, the data bits remain unmodified ($b = 1, g = f_3^1$). This can be achieved by inverting the mask for f_6^2:

$$f_{13}^2(m, d) = f_6^2(\neg m, d) = \neg m \vee d$$

4.6 Functional completeness

- See Truth Function, section Functional completeness and main article Functional completeness.
- See also “You cannot NOT have NOT” in Is XOR a combination of AND and NOT operators?.

It is established that functions NOT and AND are sufficient to generate all other binary logical functions. Since function AND can be generated with functions NOT and OR:

$$\begin{aligned} & (p \wedge q) \\ = \; & \neg(\neg(p \wedge q))) \\ = \; & \neg(\neg p \vee \neg q), \end{aligned}$$

functions NOT and OR are also functionally complete.

When looking for a single binary Boolean function that is functionally complete, candidates are all functions that generate a unary NOT: $f_i^2(p,p) = \neg p$:

			f_8^2	f_{10}^2	f_{12}^2	f_{14}^2
p	q		$\downarrow$	$\neg q$	$\neg p$	$\uparrow$
0	0		1	1	1	1
1	1		0	0	0	0

this identifies f_8^2 (NOR), f_{10}^2 ($\neg q$), f_{12}^2 ($\neg p$), f_{14}^2 (NAND).

Functions f_{10}^2 and f_{12}^2 are actually unary functions, each ignoring one of the inputs while negating the other. Since they are equivalent to a unary negation, they cannot be functionally complete, as NOT by itself is not functionally complete.

			f_{10}^2	f_{12}^2
p	q		$\neg q$	$\neg p$
0	0		1	1
0	1		0	1
1	0		1	0
1	1		0	0

However, functions f_8^2 (NOR) and f_{14}^2 (NAND) are already very close to the functions AND and OR. Both provide NOT via $f_8^2(p,p) = f_{14}^2(p,p) = \neg p$ as shown above.

			f_8^2	f_{14}^2
p	q		$\downarrow$	$\uparrow$
0	0		1	1
0	1		0	1
1	0		0	1
1	1		0	0

f_8^2 (NOR) can be used to express f_7^2 (OR) as:

$$\begin{aligned} \neg p &= p \downarrow p \\ p \vee q &= \neg(p \downarrow q) \\ p \vee q &= (p \downarrow q) \downarrow (p \downarrow q), \end{aligned}$$

which makes f_8^2 functionally complete.

f_{14}^2 (NAND) can be used to express f_1^2 (AND) as:

$$\begin{aligned} \neg p &= p \uparrow p \\ p \wedge q &= \neg(p \uparrow q) \\ p \wedge q &= (p \uparrow q) \uparrow (p \uparrow q), \end{aligned}$$

which makes f_{14}^2 functionally complete.

For $\{\mathrm{AND}, \mathrm{XOR}, \top\}$, negation can be expressed as:

$$\neg p = \top \veebar p.$$

The resulting set of functions $\{\mathrm{AND}, \mathrm{XOR}, \top, \mathrm{NOT}\}$ is a superset of the functionally complete set $\{\mathrm{AND}, \mathrm{NOT}\}$, and therefore also functionally complete.

4.6.1 Composition of Boolean functions with NOT and AND

Proposition: All Boolean functions in $F^n, n \in \mathbb{N}_0$ can be composed from Boolean functions f_2^1 aka (NOT , $\neg$) and f_1^2 aka (AND , $\wedge$).

Composing nullary functions from NOT and AND

Article Truth Function argues that only functions in $F^m, m \in \mathbb{N}$ can be composed.

However, the nullary function f_0^0 aka (FALSE , F , $\bot$) is quite obviously equivalent to all functions f_0^m for all combinations of input values $((x_k)_j)$:

$$f_0^m(x_0, x_1, \ldots, x_{m-1}) = f_0^{m-1}(x_0, x_1, \ldots, x_{m-2}).$$

So any function f_0^m with arbitrary input values can always be reduced to f_0^0:

$$f_0^2(p, q) = f_0^1(p) = f_0^0().$$

When trying to expand a function f_0^n to f_0^{n+1}, the problem arises, where the input variables should come from:

$$f_0^0() = f_0^1(\ldots) = f_0^2(\ldots).$$

An obvious choice would be constant values, which avoids inventing arbitrary variables:

$$f_0^0() = f_0^1(0) = f_0^2(0, 0).$$

Another possibility is using a variable symbol like ϕ, that is not generally used:

$$f_0^0() = f_0^1(\phi) = f_0^2(\phi, \phi),$$

which makes all functions f_0^n equivalent:

$$f_0^n(x_0, x_1, \ldots, x_{n-1}) = f_0^{n+1}(x_0, x_1, \ldots, x_{n-1}, \phi).$$

Therefore f_0^0 is proved expressable by a set of functions if any representative f_0^n can be composed.

f_0^1 and f_0^2 can be defined by NOT and AND in various ways, which are all equivalent t0 f_0^0:

$$\begin{array}{lll}
f_0^1(p) & = f_1^2(p, f_2^1(p)) & = p \wedge \neg p, \\
f_0^2(p, q) & = f_1^2(f_1^2(p, q), f_2^1(f_1^2(p, q))) & = (p \wedge q) \wedge \neg(p \wedge q), \\
 & = f_0^1(f_1^2(p, q)) & = (p \wedge q) \wedge \neg(p \wedge q), \\
 & = f_1^2(f_0^1(p), f_0^1(q)) & = (p \wedge \neg p) \wedge (q \wedge \neg q), \\
 & = f_1^2(p, f_0^1(q)) & = p \wedge (q \wedge \neg q), \\
 & = f_1^2(f_0^1(p), q) & = (p \wedge \neg p) \wedge q.
\end{array}$$

The expression

$$f_1^2(\phi, f_2^1(\phi))$$

is used to expand f_0^0. This ensures that the full expansion of a formula contains only the functions f_2^1 and f_1^2, formally satisfying functional completeness.

Trivially the nullary function f_1^0 aka (TRUE , T , $\top$) is defined as negation of f_0^0:

$$f_1^0 = f_2^1(f_0^0()) = \neg\bot = \top.$$

Convenience definition of OR

It is convenient to use f_7^2 aka (OR , $\vee$) in addition to NOT and AND, so this is proved first by induction over the complete truth table:

$$p \vee q = \neg(\neg(p \vee q)) = \neg(\neg p \wedge \neg q)$$
$$f_7^2(p,q) = f_2^1(f_1^2(f_2^1(p), f_2^1(q)))$$

p	q		$\neg p$	$\neg q$	$\neg p \wedge \neg q$	$\neg(\neg p \wedge \neg q)$	$f_7^2(p,q), \vee$
0	0		1	1	1	0	0
0	1		1	0	0	1	1
1	0		0	1	0	1	1
1	1		0	0	0	1	1

4.6.2 Universal composition function

It is obvious, that a truth table for a Boolean function $f_i^m(x_0, x_1, \ldots, x_{m-1})$ can be partitioned into an upper half, where $x_0 = 0$ and a lower half, where $x_0 = 1$. Each half without x_0 is then a truth table for a function $f_u^{m-1}(x_1, \ldots, x_{m-1})$ and a function $f_v^{m-1}(x_1, \ldots, x_{m-1})$ of arity $m-1$, $u, v \in (0, 1, \ldots, 2^{2^{m-1}} - 1), z \in (0, 1, \ldots, 2^{2^m} - 1)$. E.g.:

j	X_0	X_1	X_2	Y_{23}
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

The composition function for $m > 1$

$$c_i^m : B^m \rightarrow B^1, c_i^m(x_0, x_1, \ldots, x_k) = \begin{cases} f_u^{m-1}(x_1, \ldots, x_k) & \text{if } x_0 = 0 \\ f_v^{m-1}(x_1, \ldots, x_k) & \text{if } x_0 = 1, \end{cases}$$

and the composition function for $m = 1$

$$c_z^1 : B \rightarrow B, c_z^1(x_0) = \begin{cases} f_u^0() = Y_{z_0} & \text{if } x_0 = 0 \\ f_v^0() = Y_{z_1} & \text{if } x_0 = 1, \end{cases}$$

can be expressed in terms of logical operators $\{\bot, \top, \mathrm{NOT}, \mathrm{AND}, \mathrm{OR}\}$:

$$\begin{aligned} c_i^m(x_0, x_1, \ldots, x_k) &= (\neg x_0 \wedge f_u^{m-1}(x_1, \ldots, x_k)) \vee (x_0 \wedge f_v^{m-1}(x_1, \ldots, x_k)), \\ c_z^1(x_0) &= (\neg x_0 \wedge f_u^0()) \vee (x_0 \wedge f_v^0()). \end{aligned}$$

This is equivalent to the functional notation

$$\begin{aligned} c_i^m(x_0, x_1, \ldots, x_k) &= f_7^2(f_1^2(f_2^1(x_0), f_u^{m-1}(x_1, \ldots, x_k)), f_1^2(x_0, f_v^{m-1}(x_1, \ldots, x_k))), \\ c_z^1(x_0) &= f_7^2(f_1^2(f_2^1(x_0), f_u^0()), f_1^2(x_0, f_v^0()). \end{aligned}$$

Therefore the composition mechanism c_i^m can be applied recursively to define any required Boolean function. The recursion is guaranteed to terminate at a function c_z^1, which only uses the previously defined functions f_0^0 aka (FALSE , F , $\bot$), f_1^0 aka (TRUE , T , $\top$), f_2^1 aka (NOT , $\neg$), f_1^2 aka (AND , $\wedge$), f_7^2 aka (OR , $\vee$).

Unary functions defined by composition function

The unary funtions f_i^1 for example, are such defined as:

$$\begin{aligned} f_0^1(p) &= f_7^2(f_1^2(f_2^1(p), f_0^0()), f_1^2(p, f_0^0())), \\ f_1^1(p) &= f_7^2(f_1^2(f_2^1(p), f_0^0()), f_1^2(p, f_1^0())), \\ f_3^1(p) &= f_7^2(f_1^2(f_2^1(p), f_1^0()), f_1^2(p, f_1^0())). \end{aligned}$$

Substituting $\bot$ for $f_0^0()$ and $\top$ for $f_1^0()$ gives:

$$\begin{aligned} f_0^1(p) &= f_7^2(f_1^2(f_2^1(p), \bot), f_1^2(p, \bot)), \\ f_1^1(p) &= f_7^2(f_1^2(f_2^1(p), \bot), f_1^2(p, \top)), \\ f_3^1(p) &= f_7^2(f_1^2(f_2^1(p), \top), f_1^2(p, \top)). \end{aligned}$$

Applying $p \wedge \bot = \bot$ and $p \wedge \top = p$ gives:

$$\begin{aligned} f_0^1(p) &= f_7^2(\bot, \bot), \\ f_1^1(p) &= f_7^2(\bot, p), \\ f_3^1(p) &= f_7^2(f_2^1(p), p). \end{aligned}$$

Finally applying $p \vee \bot = p$ and $p \vee \neg p = \top$ gives:

$$\begin{aligned} f_0^1(p) &= \bot \\ f_1^1(p) &= p \\ f_3^1(p) &= \top. \end{aligned}$$

Example composition function for f_{23}^3

$$\begin{aligned} c_{23}^3(x_0, x_1, x_2) = f_7^2(&f_1^2(f_2^1(x_0), f_7^2(f_1^2(f_2^1(x_1), f_7^2(f_1^2(f_2^1(x_2), f_0^0()), f_1^2(x_2, f_0^0()))), f_1^2(x_1, f_7^2(f_1^2(f_2^1(x_2), f_0^0()), f_1^2(x_2, f_1^0())))) \\ &f_1^2(x_0, f_7^2(f_1^2(f_2^1(x_1), f_7^2(f_1^2(f_2^1(x_2), f_0^0()), f_1^2(x_2, f_1^0()))), f_1^2(x_1, f_7^2(f_1^2(f_2^1(x_2), f_1^0()), f_1^2(x_2, f_1^0())))))). \end{aligned}$$

Optimized:

$$\begin{aligned} c_{23}^3(x_0, x_1, x_2) &= f_7^2(& f_1^2(f_2^1(x_0), f_7^2(f_1^2(f_2^1(x_1), f_0^0()), f_1^2(x_1, x_2))), && \mid \text{subst } \bot, \top \\ & & f_1^2(x_0, f_7^2(f_1^2(f_2^1(x_1), x_2), f_1^2(x_1, f_1^0())))) \\ &= f_7^2(& f_1^2(f_2^1(x_0), f_7^2(f_1^2(f_2^1(x_1), \bot), f_1^2(x_1, x_2))), && \mid p \wedge \bot = \bot, p \wedge \top = p \\ & & f_1^2(x_0, f_7^2(f_1^2(f_2^1(x_1), x_2), f_1^2(x_1, \top)))) && \mid p \vee \bot = p, p \vee \top = \top \\ &= f_7^2(& f_1^2(f_2^1(x_0), f_1^2(x_1, x_2)), && \mid \text{subst } \wedge, \vee \\ & & f_1^2(x_0, f_7^2(f_1^2(f_2^1(x_1), x_2), x_1))) \\ &= & (\neg x_0 \wedge x_1 \wedge x_2) \vee && \mid \text{normalize} \\ & & (x_0 \wedge ((\neg x_1 \wedge x_2) \vee x_1)) \\ &= & (\neg x_0 \wedge x_1 \wedge x_2) \vee \\ & & (x_0 \wedge \neg x_1 \wedge x_2) \vee \\ & & (x_0 \wedge x_1). \end{aligned}$$

CHAPTER

FIVE

SOURCE CONTROL MANAGEMENT

5.1 Kallithea

Kallithea is a Python software providing github functionality, forks, pull requests. Source line annotations.

5.2 Mercurial Repository Trimming

See CommunicatingChanges - Mercurial for descriptions of import/export, bundle/unbundle.

Use revsets for specifying a range of revisions (see revsets - Mercurial).

Implicit range endpoints of `--rev` option for bundle, etc. follow the DAG algorithm (ancestors) of `x..y` and `x::y` ranges. Use `x:y` to get all revisions between x and y, regardless of ancestry.

```
mkdir -p ../repo-trimmed
hg bundle --base null --rev 0:240 ../repo-trimmed/bundle.hg
( cd ../repo-trimmed && hg init && hg unbundle bundle.hg)
```

When using `hg convert`, it is necessary to use the configuration parameter *convert.hg.revs*, since option *–ref* does not understand revsets:

> hg convert –config convert.hg.revs=0:240 repo repo-trimmed

Using **`hg export`** and **`hg import`**, it is possilbe to cherry-pick changesets and move them to branches, etc.

```
cd repo
hg export --rev '262:263' >.00-merge.patch

cd ../repo-trimmed
hg import ../repo/00-merge.patch
```

Cherry-pick some changesets, move them on a branch and tag the result:

```
cd repo
hg export --rev '241 or 243:259 or 261' >01-branch-abstract-diagrams.patch

cd ../repo-trimmed
hg branch abstract-diagrams
hg import ../repo/01-branch-abstract-diagrams.patch
hg tag diag-1-too-abstract
```

CHAPTER

SIX

SHELL BASICS

6.1 Job Control

Command	Description
`jobs`	show jobs (background tasks)
`C-z`	send signal TSTOP to foreground job, which stops, is sent to the background job queue and becomes the current job
`bg [job no.]`	run current job in job queue in background
`fg [job no.]`	run current job in foreground
`cmd &`	run cmd as background job

6.2 POSIX

Using POSIX compatible syntax allows shell scripts to run on other systems, where no bash(1) is available (NAS, various embedded systems, old systems with proprietary unixes).

So, use *#!/bin/sh* instead of *#!/bin/bash*.

Use The Open Group Base Specifications Issue 7, 2018 edition for general reference. See sh - shell, the standard command language interpreter and Shell Command Language for specific shell reference.

> **Attention:** Be aware that there are very old shells out there that do not conform to POSIX. Mainly because POSIX was not around at their conception. So not everything allowed by POSIX is necessarily failsafe for all shells.

6.2.1 Conventions for Syntax Descriptions

The conventions for syntax descriptions are covered in Chapter 12. Utility Conventions of The Open Group Base Specifications Issue 7, 2018 edition.

An informal description can also be found in man-pages(7).

The syntax for alternative arguments `{ arg1 | arg2 | arg3 }` is not part of POSIX, but is mentioned in syntax - Is there a specification for a man page's SYNOPSIS section? - Stack Overflow

Specific Conventions in Templated Shell Scripts

Single letter options preceded by a single dash `-` must not be grouped together. The additional effort is not worth the conceived advantage. Readability is much better, when separating single letter options.

In addition to POSIX chapter 12, a shortened ellipses `..` may be used in place of a full ellipses `...`.

Clarifying POSIX section 12.8, the vertical bar `|` is only used within braces `{`, `}` to indicate exclusive alternatives. Together with brackets `[`, `]` This allows specifying optional syntax variants without an implied order. E.g.:

```
program [ { key=[value] | -key | [!]key } ..]
```

This notation emphasizes that each of these options overrides the effect of a previous occurence. The following syntax description is equivalent, but the exclusive nature of the alternatives is not so clear:

```
program [key=[value]].. [-key].. [[!]key]..
```

This example can be further shortened, if the option description mentions, that the option can be specified multiple times:

```
program [key=[value]] [-key] [[!]key]
```

In addtion to specifying multiple synopsis lines according to POSIX section 12.8, mutually exclusive option may be given summarily as [MODE OPTIONS], which are understood to override any previous mode options. E.g.:

```
  program [OPTIONS] [MODE OPTIONS]

MODE OPTIONS
  --one  perform action 1
  --two  perform action 2
```

6.3 Special Purpose Language vs. Generic Programming Language

The bourne shell sh(1) is a special purpose language. It is **not** a generic programming language.

Making the shell more like C, with e.g. csh(1), are misguided experiments.

Making the excution of the **`test`** program look like a condition in a programming language is a very special brain dead example of syntactic obfuscation.

The standard syntax shows quite clearly, what happens, when the program **`test`** is executed:

```
if test arg1 arg2
then
    :
fi
```

The alternate program name **`[`** requires an extra argument *]* for *closing* the fake opening bracket, just so the command execution resembles a *mathematical* condition:

```
if [ arg1 arg2 ]
then
    :
fi
```

Warning: Using this abomination in a shell scripts results in immediate deletion.

Avoid arithmetic expansion `$(( ... ))`, if **`expr`**(1) can do the job.

6.3.1 Variable expansion

To be safe and to make replacemnts simpler, always use curly braces for variable expansion:

```
printf "variable: %s, arg count: %d, args: %s\n" "${variable}" "${#}" "${*}"
```

Emacs support in Shell-script mode:

Shortcut	Expansion
`C-c v`	${}
`C-c q`	"${}"

6.3.2 echo (1) , printf(1)

Do not use echo(1), since it is not portable, use printf(1) instead.

Emacs support in Shell-script mode:

Shortcut	Expansion
`C-c p`	printf "%sn"
`C-u C-c p`	printf >&2 "%sn"

Especially dash(1) (ubuntu system shell) and bash(1) differ extremely.

```
$ ls -l /bin/sh
lrwxrwxrwx 1 root root 4 Mai  8  2018 /bin/sh -> dash

$ /bin/dash -c 'echo "hello\nnext line"'
hello
next line

$ /bin/dash -c 'echo -e "hello\nnext line"'
-e hello
next line

$ /bin/bash -c 'echo "hello\nnext line"'
hello\nnext line

$ /bin/bash -c 'echo -e "hello\nnext line"'
hello
next line
```

Emacs support in Shell-script mode for debug output of variables:

`arg_count C-c d v v`

expands to

```
printf >&2 "#    "":DBG:   %-${dbg_fwid-15}s: [%s]\n" "arg_count" "${arg_count}"
```

6.3.3 Avoid special bash syntax

Do not use:

```
function func_name
{
    :
}
```

but use POSIX compatible syntax instead:

```
func_name ()
{
    :
}
```

6.3.4 Do not use arrays

Shell arrays are not POSIX compatible! If you think you need to use arrays, you should probably not use the shell but a generic script programming language like awk(1), perl(1) or python(1).

See section 6.4, WRF loop - single line processing in shell for single line processing with splitting into fields.

See also bash - How to mark an array in POSIX sh? - Stack Overflow

See also GitHub - krebs/array: a POSIX-compliant implementation of arrays, for a POSIX compliant implementation of arrays (untested).

6.4 WRF loop - single line processing in shell

Emulating single line processing like sed(1) and awk(1) with **read** in a **while** loop. WRF stands historically for while/read/file.

6.4.1 WRF loop

A file is parsed as single lines with the **read** command (see listing 6.1, line 14):

```
while read -r in_line
```

See figure 6.1 for activity diagram.

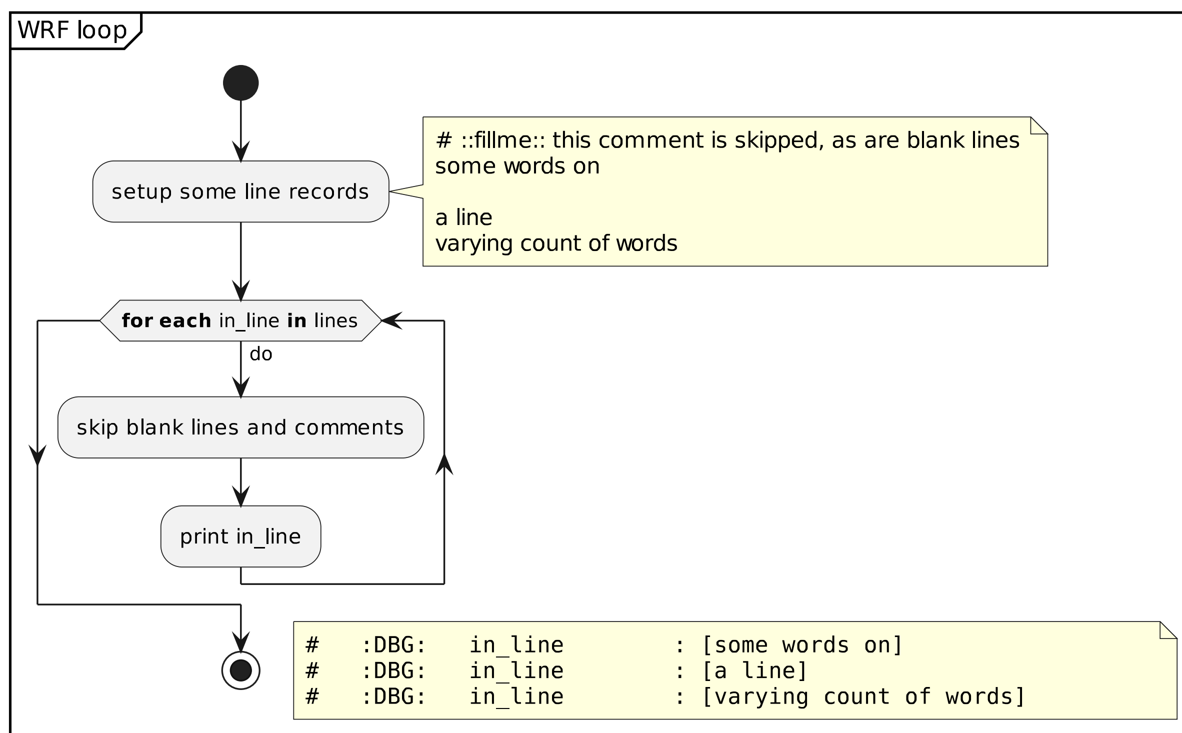


figure 6.1: WRF loop

listing 6.1: WRF loop

```
1 # setup some line records
2 in_records="
3 # ::fillme:: this comment is skipped, as are blank lines
4 some words on
```

(continues on next page)

(continued from previous page)

```
5
6   a line
7   varying count of words
8   "
9
10  printf "%s\n" "${in_records}" \
11  | (
12  while read -r in_line
13  do
14      # skip blank lines and comments
15      case "${in_line}" in
16      ''|"${comm-#}"*) continue;;
17      esac
18
19      # print in_line
20      printf >&2 "#    "":DBG:   %-${dbg_fwid-15}s: [%s]\n" "in_line" "${in_line}"
21  done
22  )
```

6.4.2 WRF loop with standard IFS split

Instead of reading an entire line, the **read** command parses the line into several variables (see listing 6.2, line 14):

```
while read -r in_word0 in_word1 rest
```

The standard IFS is used which splits the line on whitespace.

See figure 6.2 for activity diagram.

listing 6.2: WRF loop with standard IFS split

```
1   # setup some line records with
2   # standard IFS whitespace separator
3   in_records="
4   # ::fillme:: this comment is skipped, as are blank lines
5   some words on
6
7   a line
8   varying count of words
9   "
10
11  printf "%s\n" "${in_records}" \
12  | (
13  # use standard IFS to split line
14  while read -r in_word0 in_word1 rest
15  do
16      # skip blank lines and comments
17      case "${in_word0}" in
18      ''|"${comm-#}"*) continue;;
19      esac
20
21      # print parts
22      printf >&2 "# --------------------------------------------------\n"
23      printf >&2 "#    "":DBG:   %-${dbg_fwid-15}s: [%s]\n" "in_word0" "${in_word0}"
24      printf >&2 "#    "":DBG:   %-${dbg_fwid-15}s: [%s]\n" "in_word1" "${in_word1}"
25      printf >&2 "#    "":DBG:   %-${dbg_fwid-15}s: [%s]\n" "rest" "${rest}"
26  done
27  )
```

6.4.3 WRF loop with special IFS split

Instead of reading an entire line, the **read** command parses the line into several variables (see listing 6.3, line 14):

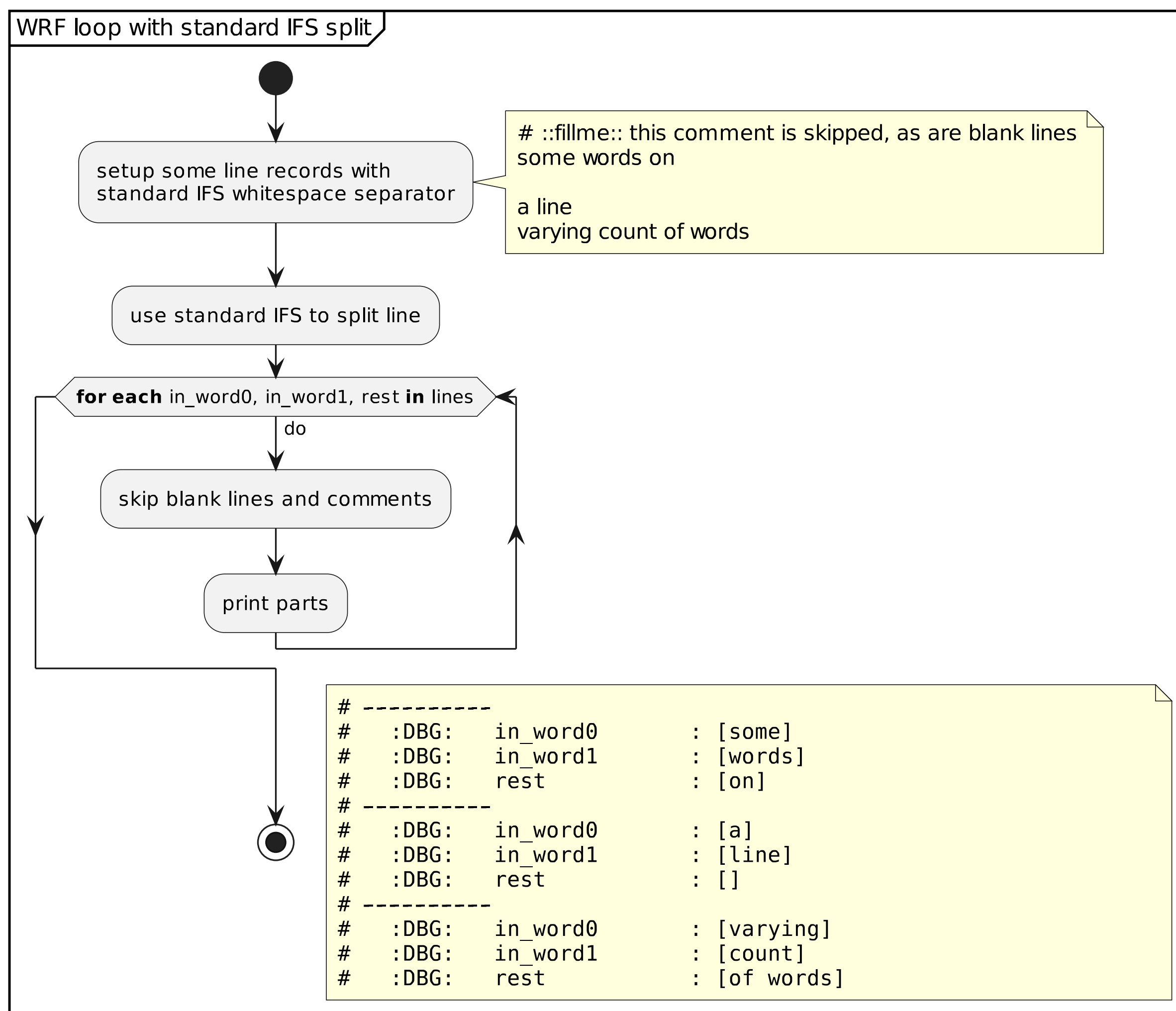


figure 6.2: WRF loop with standard IFS split

```
while IFS=: read -r in_word0 in_word1 rest
```

IFS is set to `:` for the **read** command only, which splits the line on a `:` character.

See figure 6.3 for activity diagram.

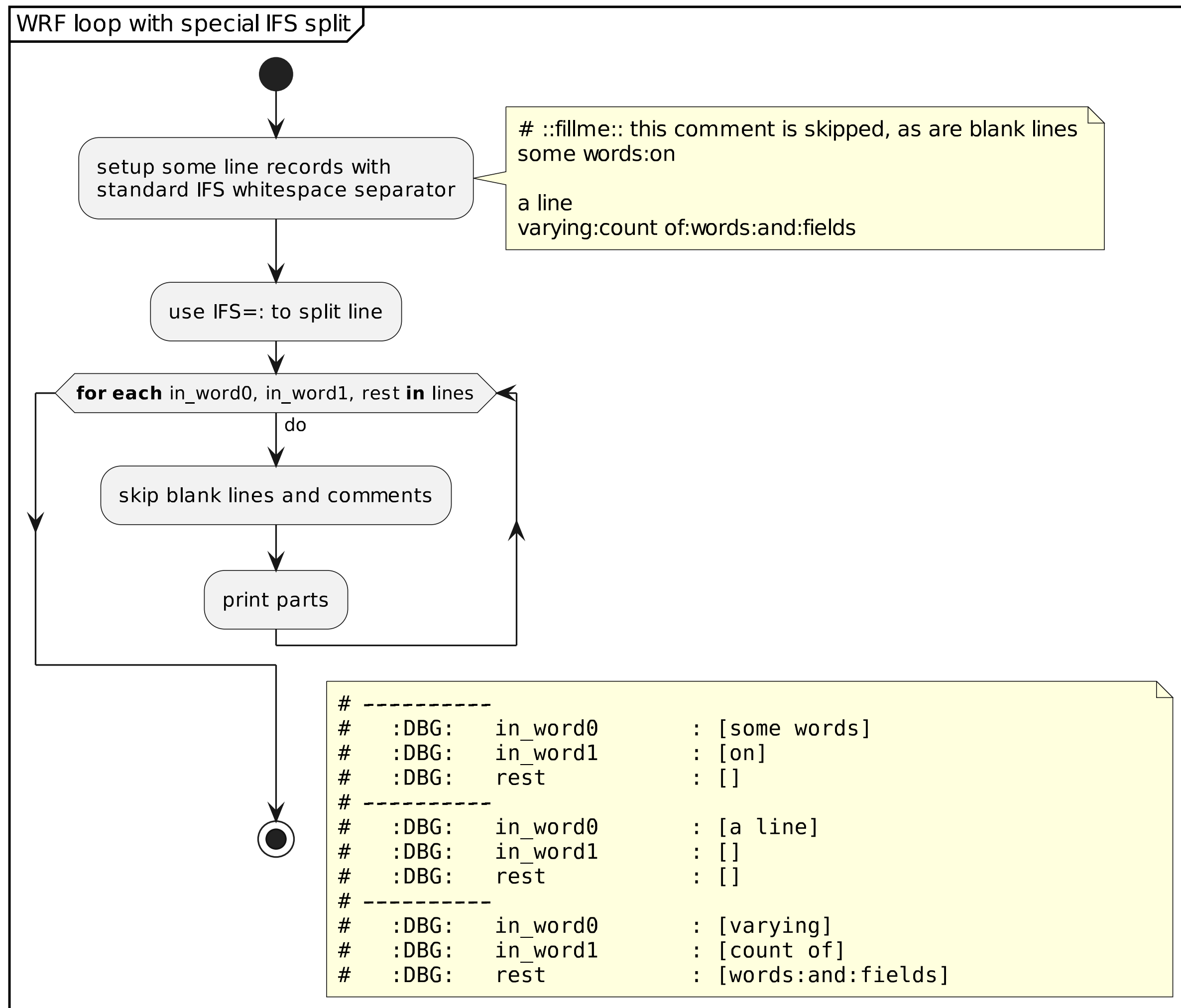


figure 6.3: WRF loop with special IFS split

listing 6.3: WRF loop with special IFS split

```
 1 # setup some line records with
 2 # standard IFS whitespace separator
 3 in_records="
 4 # ::fillme:: this comment is skipped, as are blank lines
 5 some words:on
 6
 7 a line
 8 varying:count of:words:and:fields
 9 "
10
11 printf "%s\n" "${in_records}" \
12 | (
13 # use IFS=: to split line
14 while IFS=: read -r in_word0 in_word1 rest
15 do
16     # skip blank lines and comments
17     case "${in_word0}" in
18     ''|"${comm-#}"*) continue;;
```

(continues on next page)

(continued from previous page)

```
    esac

    # print parts
    printf >&2 "# --------------------------------------------------\n"
    printf >&2 "#   "":DBG:   %-${dbg_fwid-15}s: [%s]\n" "in_word0" "${in_word0}"
    printf >&2 "#   "":DBG:   %-${dbg_fwid-15}s: [%s]\n" "in_word1" "${in_word1}"
    printf >&2 "#   "":DBG:   %-${dbg_fwid-15}s: [%s]\n" "rest" "${rest}"
done
)
```

6.4.4 split and process lines with awk(1)

listing 6.4: AWK script to split and process lines via callback

```
BEGIN {
    if (!line) {
        line = "line";
    }
    if (!varbase) {
        varbase = "col";
    }
    if (!callback) {
         callback = "col_process";
    }
    if (!max_count) {
         max_count = 10;
    }
}
function single_quote_enclose (str) {
    gsub(/'/, "'\\''", str);
    return "'" str "'";
}
{
    printf("%s=%s;\n", line, single_quote_enclose($0));
    printf("%s_count=%d;\n", varbase, NF);
    for (_i=1;_i<=NF;++_i) {
        printf("%s%d=%s;\n", varbase, _i, single_quote_enclose($_i));
    }
    for (_i=NF+1;_i<=max_count;++_i) {
        printf("%s%d=;\n", varbase, _i);
    }
    if (callback) {
        printf("%s;\n\n", callback);
    }
}
```

listing 6.5: Function split and process lines via callback

```
#!   split_and_process_lines [-F " *: *"] [-v line=line] [-v varbase=col] [-v callback=col_
↪process] [-v max_count=10]
split_and_process_lines ()                                                  # ||:fnc:||
{
    ${AWK__PROG-awk} ${1+"$@"} "${AWK_SCRIPT_SPLIT_AND_PROCESS_LINES}"
}
```

listing 6.6: Example for split and process lines via callback

```
in_records="
# comment
in the : city
and : over : the : mountains

som'e where : over ' the : rainbow
some wh''ere : over the : rainbow
"

```

(continues on next page)

(continued from previous page)

```
10  col_process ()
11  {
12  case "${line}" in
13  '#'*|'')
14      printf >&2 "#  |"":WRN:|  warning: comment or blank line [%s]\n" "${line}"
15      return
16      ;;
17  esac
18
19  printf "line: col1 [%s] col2 [%s] col3 [%s] col4 [%s] col5 [%s]\n" "${col1}" "${col2}" "$
    ↪{col3}" "${col4}" "${col5}"
20
21  _indx=1
22  while test ${_indx} -le ${col_count}
23  do
24      _var="col${_indx}"
25      eval _value=\"\${${_var}}\"
26
27      printf "      %s=%s\n" "${_var}" "${_value}"
28
29      _indx="$( expr ${_indx} + 1 )"
30  done
31  }
32
33  _script="$(
34    printf "%s\n" "${in_records}" \
35    | split_and_process_lines -F ' *: *' -v varbase='col' -v callback='col_process'
36    )"
37
38  printf >&2 "# --------------------------------------------------\n"
39  printf >&2 "#   "":DBG:   %-${dbg_fwid-15}s: [%s]\n" "_script" "${_script}"
40
41  printf >&2 "# --------------------------------------------------\n"
42  eval "${_script}"
```

listing 6.7: Script generated by example for split and process lines via callback

```
# --------------------------------------------------
#   :DBG:   _script        : [line='';
col_count=0;
col1=;
col2=;
col3=;
col4=;
col5=;
col6=;
col7=;
col8=;
col9=;
col10=;
col_process;

line='# comment';
col_count=1;
col1='# comment';
# ...
col_process;

line='in the : city';
col_count=2;
col1='in the';
col2='city';
# ...
col_process;

line='and : over : the : mountains';
col_count=4;
col1='and';
col2='over';
```

(continues on next page)

(continued from previous page)

```
col3='the';
col4='mountains';
# ...
col_process;

line='';
col_count=0;
# ...
col_process;

line='som'\''e where : over '\'' the : rainbow';
col_count=3;
col1='som'\''e where';
col2='over '\'' the';
col3='rainbow';
# ...
col_process;

line='some wh'\'''\''ere : over the : rainbow';
col_count=3;
col1='some wh'\'''\''ere';
col2='over the';
col3='rainbow';
# ...
col_process;

line='';
col_count=0;
# ...
col_process;]
```

listing 6.8: Output from example for split and process lines via callback

```
# --------------------------------------------------
#   |:WRN:|  warning: comment or blank line []
#   |:WRN:|  warning: comment or blank line [# comment]
line: col1 [in the] col2 [city] col3 [] col4 [] col5 []
      col1=in the
      col2=city
line: col1 [and] col2 [over] col3 [the] col4 [mountains] col5 []
      col1=and
      col2=over
      col3=the
      col4=mountains
#   |:WRN:|  warning: comment or blank line []
line: col1 [som'e where] col2 [over ' the] col3 [rainbow] col4 [] col5 []
      col1=som'e where
      col2=over ' the
      col3=rainbow
line: col1 [some wh''ere] col2 [over the] col3 [rainbow] col4 [] col5 []
      col1=some wh''ere
      col2=over the
      col3=rainbow
#   |:WRN:|  warning: comment or blank line []
```

6.5 Single quoting

1. Starting with an unquoted string, assume that it is part of a single quoted string already:

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
e	m	b	e	d	'	e	d	'	s	i	n	g	'	l	e											

2. Find next embedded single quote and insert a single quote before to terminate single quoting `'` -> `''`:

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
					↓																					
e	m	b	e	d	'	'	e	d	'	s	i	n	g	'	l	e										

3. Insert a backslash to escape the embedded single quote `'` -> `'\'`:

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
						↓																				
e	m	b	e	d	'	\	'	e	d	'	s	i	n	g	'	l	e									

4. Add a single quote after the escaped single quote to continue quoting `'` -> `'\''`:

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
								↓																		
e	m	b	e	d	'	\	'	'	e	d	'	s	i	n	g	'	l	e								

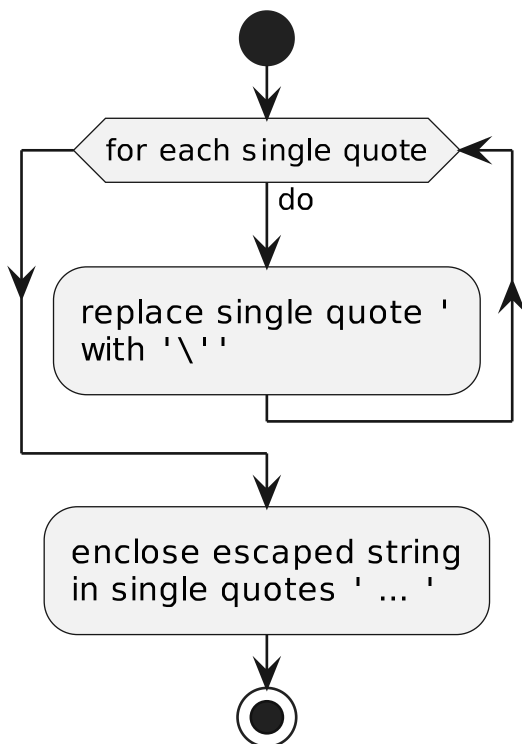
5. Repeat steps 2 through 4 for all remaining embedded single quotes `'` -> `'\''`:

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
											↓	↓		↓					↓	↓		↓				
e	m	b	e	d	'	\	'	'	e	d	'	\	'	'	s	i	n	g	'	\	'	'	l	e		

6. Enclose string in single quotes `' ... '` to complete single quoting:

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
↓																										↓
'	e	m	b	e	d	'	\	'	'	e	d	'	\	'	'	s	i	n	g	'	\	'	'	l	e	'

Activity diagram for algorithm:



6.6 Construct correctly quoted shell script

A shell script is assigned to the variable *_script* to be executed for different purposes, e.g.

- at a later time:

```
eval "${_script}"
```

- in different shell process, e.g.:

```
sh -c "${_script}"
printf "%s\n" "${_script}" | sh
```

- as different user:

```
sudo -u user sh -c "${_script}"
```

- on remote host:

```
ssh user@host "${_script}"
printf "%s\n" "${_script}" | ssh user@host
```

6.6.1 Preparations

1. Update snippets to latest version:

```
cd /srv/ftp/pub && ./sync.sh --restore && ./xx-sync-ftp-pub.sh -l 0
```

2. Create test shell script with template:

```
snn x_quoted_script.sh
```

3. Expand snippet (at end of line (`C-e`) enter key sequence `C-x C-e`):

```
## (progn (forward-line 1) (snip-insert "sh_f.single-quote" t t "sh" " --key single_
↪quote_minimal") (insert "\n"))
```

4. Add example environment setup in body:

```
set -- arg1 arg2 'arg with spaces'

TEMP_DIR='/tmp/some-rndajom-stuff'
```

5. Add example command:

```
( cd "${TEMP_DIR}/" || exit 1; pwd )
for _arg in ${1+"${@}"}; do echo "${_arg}"; done
echo 'hello' | cat -
```

6. Execute and study output:

```
x_quoted_script.sh: 2: cd: cannot cd to /tmp/some-rndajom-stuff/
arg1
arg2
arg with spaces
hello
```

6.6.2 Single quoted string

1. Single quote entire command:

```
_script='
( cd "${TEMP_DIR}/" || exit 1; pwd )
for _arg in ${1+"${@}"}; do echo "${_arg}"; done
echo '\''hello'\'' | cat -
'
```

add some execution tests:

```
printf "%s\n" "${_script}"

printf "%s\n" '--------------------------------------------------'
eval "${_script}"

printf "%s\n" '--------------------------------------------------'
sh -c "${_script}"
```

and observe output:

```
( cd "${TEMP_DIR}/" || exit 1; pwd )
for _arg in ${1+"${@}"}; do echo "${_arg}"; done
echo 'hello' | cat -
--------------------------------------------------
x_quoted_script.sh: 2: cd: cannot cd to /tmp/some-rndajom-stuff/
arg1
arg2
arg with spaces
hello
--------------------------------------------------
/
hello
```

2. Interrupt quoting to insert expanded variables.

   1. Use `single_quote_enclose()` as necessary:

   ```
   _script='
   ( cd '"$( single_quote_enclose "${TEMP_DIR}/" )"' || exit 1; pwd )
   for _arg in ${1+"${@}"}; do echo "${_arg}"; done
   echo '\''hello'\'' | cat
   '
   ```

   2. Use `single_quote_args()` as necessary:

   ```
   _script='
   ( cd '"$( single_quote_enclose "${TEMP_DIR}/" )"' || exit 1; pwd )
   for _arg in '"$( single_quote_args ${1+"${@}"} )"'; do echo "${_arg}"; done
   echo '\''hello'\'' | cat -
   '
   ```

and observe output:

```
( cd '/tmp/some-rndajom-stuff/' || exit 1; pwd )
for _arg in 'arg1' 'arg2' 'arg with spaces'; do echo "${_arg}"; done
echo 'hello' | cat -
--------------------------------------------------
x_quoted_script.sh: 2: cd: cannot cd to /tmp/some-rndajom-stuff/
arg1
arg2
arg with spaces
hello
--------------------------------------------------
sh: 2: cd: cannot cd to /tmp/some-rndajom-stuff/
arg1
arg2
arg with spaces
hello
```

6.6.3 HERE document

1. Enclose entire command in `cat <<EOF ... EOF`, escape as necessary:

   ```
   cat <<EOF
   ( cd "\${TEMP_DIR}/" || exit 1; pwd )
   for _arg in \${1+"\${@}"}; do echo "\\\${_arg}"; done
   echo 'hello' | cat -
   EOF
   ```

   and observe output:

   ```
   ( cd "/tmp/some-rndajom-stuff/" || exit 1; pwd )
   for _arg in arg1 arg2 arg with spaces; do echo "${_arg}"; done
   echo 'hello' | cat -
   ```

2. Use `single_quote_enclose()` and `single_quote_args()` as necessary:

   ```
   cat <<EOF
   ( cd \$( single_quote_enclose "\${TEMP_DIR}/" ) || exit 1; pwd )
   for _arg in \$( single_quote_args \${1+"\${@}"} ); do echo "\\\${_arg}"; done
   echo 'hello' | cat -
   EOF
   ```

   and observe output:

   ```
   ( cd '/tmp/some-rndajom-stuff/' || exit 1; pwd )
   for _arg in 'arg1' 'arg2' 'arg with spaces'; do echo "${_arg}"; done
   echo 'hello' | cat -
   ```

3. Enclose in subshell expansion `"$( ... )"` for assignment to variable:

   ```
   _script="$(
   cat <<EOF
   ( cd \$( single_quote_enclose "\${TEMP_DIR}/" ) || exit 1; pwd )
   for _arg in \$( single_quote_args \${1+"\${@}"} ); do echo "\\\${_arg}"; done
   echo 'hello' | cat -
   EOF
   )"
   ```

   add some execution tests:

   ```
   printf "%s\n" "${_script}"

   printf "%s\n" '--------------------------------------------------'
   eval "${_script}"

   printf "%s\n" '--------------------------------------------------'
   sh -c "${_script}"
   ```

   and observe output:

   ```
   ( cd '/tmp/some-rndajom-stuff/' || exit 1; pwd )
   for _arg in 'arg1' 'arg2' 'arg with spaces'; do echo "${_arg}"; done
   echo 'hello' | cat -
   --------------------------------------------------
   x_quoted_script.sh: 1: cd: cannot cd to /tmp/some-rndajom-stuff/
   arg1
   arg2
   arg with spaces
   hello
   --------------------------------------------------
   sh: 1: cd: cannot cd to /tmp/some-rndajom-stuff/
   arg1
   arg2
   arg with spaces
   hello
   ```

4. Enclose entire command in here document specifying a quoted end-of-file marker `cat <<'EOF'... EOF`, no escaping is necessary:

```
cat <<'EOF'
( cd "${TEMP_DIR}/" || exit 1; pwd )
for _arg in ${1+"${@}"}; do echo "\${_arg}"; done
echo 'hello' | cat -
EOF
```

   and observe output:

```
( cd "${TEMP_DIR}/" || exit 1; pwd )
for _arg in ${1+"${@}"}; do echo "\${_arg}"; done
echo 'hello' | cat -
```

   The type of quotes (single or double) does not matter.

6.7 Command execution

For bash(1), four types of commands are defined:

- aliases
- shell functions
- builtin commands
- external programs

A POSIX shell like dash(1) does not support aliases.

From the man page of bash(1):

> **COMMAND EXECUTION**
>
> After a command has been split into words, if it results in a simple command and an optional list of arguments, the following actions are taken.
>
> If the command name contains no slashes, the shell attempts to locate it. [If the shell is interactive or shell option expand_aliases is set and an alias by that name is found, it is expanded.] If there exists a shell function by that name, that function is invoked as described above in FUNCTIONS. If the name does not match a function, the shell searches for it in the list of shell builtins. If a match is found, that builtin is invoked.
>
> If the name is neither a shell function nor a builtin, and contains no slashes, bash searches each element of the PATH for a directory containing an executable file by that name. Bash uses a hash table to remember the full pathnames of executable files (see hash under SHELL BUILTIN COMMANDS below). A full search of the directories in PATH is performed only if the command is not found in the hash table. If the search is unsuccessful, the shell searches for a defined shell function named command_not_found_handle. If that function exists, it is invoked with the original command and the original command's arguments as its arguments, and the function's exit status becomes the exit status of the shell. If that function is not defined, the shell prints an error message and returns an exit status of 127.
>
> If the search is successful, or if the command name contains one or more slashes, the shell executes the named program in a separate execution environment. Argument 0 is set to the name given, and the remaining arguments to the command are set to the arguments given, if any.
>
> If this execution fails because the file is not in executable format, and the file is not a directory, it is assumed to be a shell script, a file containing shell commands. A subshell is spawned to execute it. This subshell reinitializes itself, so that the effect is as if a new shell had been invoked to handle the script, with the exception that the locations of commands remembered by the parent (see hash below under SHELL BUILTIN COMMANDS) are retained by the child.

If the program is a file beginning with #!, the remainder of the first line specifies an interpreter for the program. The shell executes the specified interpreter on operating systems that do not handle this executable format themselves. The arguments to the interpreter consist of a single optional argument following the interpreter name on the first line of the program, followed by the name of the program, followed by the command arguments, if any.

Note: DO NOT set the shell option expand_aliases in scripts. Generally, **DO NOT** write bash(1) scripts. Stick to **POSIX**.

figure 6.4 shows an activity diagram for the command execution process.

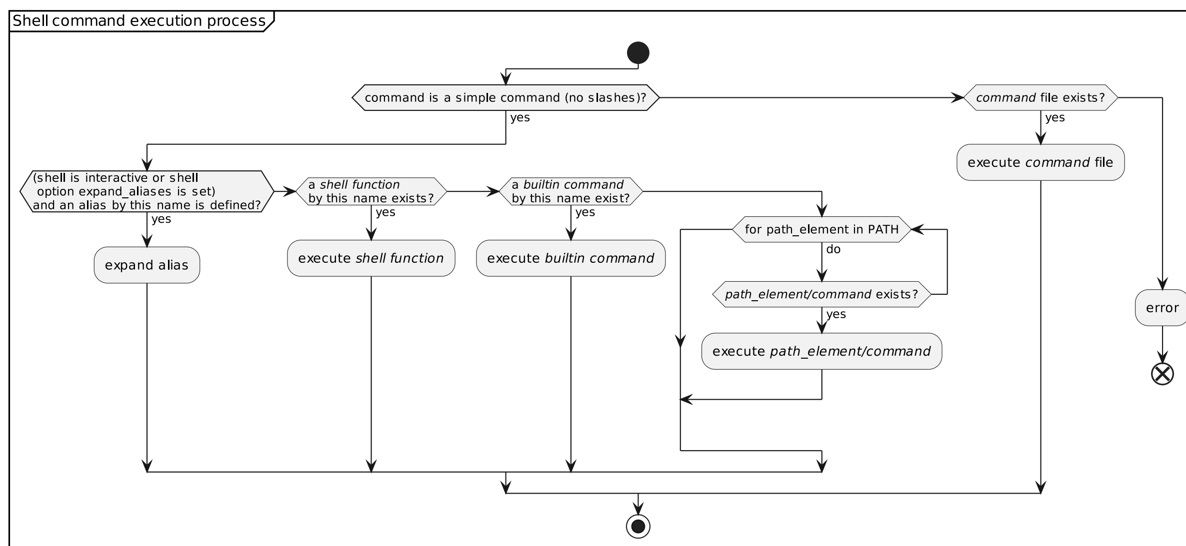


figure 6.4: Shell command execution process

6.8 . command

The **.** command is an include mechanism for script files (much like the preprocessor command *#include* in C). Note, that the standard definition of **.** ignores all arguments, which means, that no arguments are allowed to avoid inconsistent behavior for different shells..

All variable assignments in the included file are incorporated into the shell environment.

From man page of bash(1):

> **. filename [. . .] Read and execute commands from filename in the** current shell environment and return the exit status of the last command executed from filename. If filename does not contain a slash, filenames in PATH are used to find the directory containing filename. The file searched for in PATH need not be executable. When bash is not in posix mode, the current directory is searched if no file is found in PATH. If the sourcepath option to the shopt builtin command is turned off, the PATH is not searched. [. . .] The return status is the status of the last command exited within the script (0 if no commands are executed), and false if filename is not found or cannot be read.

6.9 Subshell and compound commands

From man page of bash(1):

Compound Commands A compound command is one of the following. In most cases a list in a command's description may be separated from the rest of the command by one or more newlines, and may be followed by a newline in place of a semicolon.

(list) list is executed in a subshell environment (see COMMAND EXECUTION ENVIRONMENT below). Variable assignments and builtin commands that affect the shell's environment do not remain in effect after the command completes. The return status is the exit status of list.

{ list; } list is simply executed in the current shell environment. list must be terminated with a newline or semicolon. This is known as a group command. The return status is the exit status of list. Note that unlike the metacharacters (and), { and } are reserved words and must occur where a reserved word is permitted to be recognized. Since they do not cause a word break, they must be separated from list by whitespace or another shell metacharacter.

Builtin commands in a subshell do not affect the shell environment in the parent shell, e.g.:

```
VAR='value'
( VAR='something'; echo "${VAR}"; )
echo "${VAR}";
```

results in output of:

```
something
value
```

Builtin commands in a group command do affect the shell environment outside the group, e.g.:

```
VAR='value'
{ VAR='something'; echo "${VAR}"; }
echo "${VAR}";
```

results in output of:

```
something
something
```

Note: A command in a pipeline is implicitely executed in a subshell.

I.e.:

```
var=outer
echo world | { var=inner; echo hello; cat - }
echo "${var}"
```

is equivalent to:

```
var=outer
echo world | ( var=inner; echo hello; cat - )
echo "${var}"
```

Note: Generally avoid `{ list }` grouping. Especially the side effect of shell environment manipulation.

Note: A subshell is not equivalent to execution of an external shell script.

A subshell can access all variables of the parent shell environment, whether they are exported or not. E.g.:

```
unexported='internal value'
export exported='external value'
( echo "[${unexported}]"; echo "[${exported}]" )
```

results in output

```
[internal value]
[external value]
```

Whereas an external shell script can only access exported variables of the parent shell environment. E.g.:

```
unexported='internal value'
export exported='external value'
cat <<'EOF' | sh
echo "[${unexported}]"; echo "[${exported}]"
EOF
```

results in output

```
[]
[external value]
```

CHAPTER

SEVEN

SHELL WILDCARDS AND FILENAMES

7.1 Glob pattern expansion

Glob patterns are always expanded. No restrictions are applied. See also **bin/check-glob-pattens.sh**.

```
mkdir -p glob-check
cd glob-check || exit 1
touch echo
touch file1
touch file2
touch file3
```

The command:

```
ls -l
```

shows the available files:

```
insgesamt 0
-rw-rw-r-- 1 da da 0 Okt 23 13:03 echo
-rw-rw-r-- 1 da da 0 Okt 23 13:03 file1
-rw-rw-r-- 1 da da 0 Okt 23 13:03 file2
-rw-rw-r-- 1 da da 0 Okt 23 13:03 file3
```

Command execution of:

```
echo *
```

results in:

```
echo file1 file2 file3
```

Expanding the wildcard:

```
*
```

results in:

```
echo file1 file2 file3
```

which is executed and results in:

```
file1 file2 file3
```

7.2 Non-matching glob patterns

If a glob pattern does not match, the glob pattern is used unexpanded:

```
for _file in not_here*
do
    pecho "${_file}"
done
```

Output:

```
not_here*
file1
file2
file3
```

If non-existent file should be ignored, an explicit test is necessary:

```
for _file in not_here* file*
do
    test -r "${_file}" || continue
    pecho "${_file}"
done
```

Output:

```
file1
file2
file3
```

Using ls(1) is another possibliity, (**but it does not work if filenames contain whitespace**):

```
for _file in $( ls -1 not_here* file* 2>/dev/null )
do
    pecho "${_file}"
done
```

Output:

```
file1
file2
file3
```

7.3 Directory separator and illegal characters

The directory separator is always illegal. Which character is used for directory separation depends on the filesystem.

OS	Win 95/98	Win 2000+	Mac OS 9	Mac OS X	Unix/Linux
File System	FAT 32	NTFS	HFS +	HFS+/UFS	UFS
Max Char.	255	256	31	255	255
Max Path.	260	260			
Restricted	“*/\:<>?|	“*/\:<>?|	:	/:	/
Directory Separator	\	\	:	:	/
Case Sensitivity	NO	NO	NO	NO	YES

Soure: File Systems - HFS+, UFS, FAT32 and NTFS, Operating Systems, Maximum Characters Allowed Comparison

7.3.1 7-bit ASCII, printable, without directory separator and illegal characters

7 bit can encode the characters from 0 - 127.

Table columns are: character, decimal, octal, hexadecimal, PS name, ASCII name.

Illegal characters (for any file system) are marked with the name ILLEGAL.

Printable characters starts after SPACE:

```
[ ] :  32 : 040 : x20 .
```

and end before DEL:

```
[^?] : 127 : 177 : x7F :        : DEL .
```

Printable characters are therefore

```
[ ] :  32 : 040 : x20 .
[!] :  33 : 041 : x21 .
["] :  34 : 042 : x22 :         : ILLEGAL .
[#] :  35 : 043 : x23 .
[$] :  36 : 044 : x24 .
[%] :  37 : 045 : x25 .
[&] :  38 : 046 : x26 .
['] :  39 : 047 : x27 .
[(] :  40 : 050 : x28 .
[)] :  41 : 051 : x29 .
[*] :  42 : 052 : x2A :         : ILLEGAL .
[+] :  43 : 053 : x2B .
[,] :  44 : 054 : x2C .
[-] :  45 : 055 : x2D .
[.] :  46 : 056 : x2E .
[/] :  47 : 057 : x2F :         : ILLEGAL .
[0] :  48 : 060 : x30 .
[1] :  49 : 061 : x31 .
[2] :  50 : 062 : x32 .
[3] :  51 : 063 : x33 .
[4] :  52 : 064 : x34 .
[5] :  53 : 065 : x35 .
[6] :  54 : 066 : x36 .
[7] :  55 : 067 : x37 .
[8] :  56 : 070 : x38 .
[9] :  57 : 071 : x39 .
[:] :  58 : 072 : x3A :         : ILLEGAL .
[;] :  59 : 073 : x3B .
[<] :  60 : 074 : x3C :         : ILLEGAL .
[=] :  61 : 075 : x3D .
[>] :  62 : 076 : x3E :         : ILLEGAL .
[?] :  63 : 077 : x3F :         : ILLEGAL .
[@] :  64 : 100 : x40 .
[A] :  65 : 101 : x41 .
[B] :  66 : 102 : x42 .
[C] :  67 : 103 : x43 .
[D] :  68 : 104 : x44 .
[E] :  69 : 105 : x45 .
[F] :  70 : 106 : x46 .
[G] :  71 : 107 : x47 .
[H] :  72 : 110 : x48 .
[I] :  73 : 111 : x49 .
[J] :  74 : 112 : x4A .
[K] :  75 : 113 : x4B .
[L] :  76 : 114 : x4C .
[M] :  77 : 115 : x4D .
[N] :  78 : 116 : x4E .
[O] :  79 : 117 : x4F .
[P] :  80 : 120 : x50 .
[Q] :  81 : 121 : x51 .
[R] :  82 : 122 : x52 .
[S] :  83 : 123 : x53 .
[T] :  84 : 124 : x54 .
[U] :  85 : 125 : x55 .
[V] :  86 : 126 : x56 .
[W] :  87 : 127 : x57 .
[X] :  88 : 130 : x58 .
[Y] :  89 : 131 : x59 .
[Z] :  90 : 132 : x5A .
```

(continues on next page)

(continued from previous page)

```
[[] :  91 : 133 : x5B .
[\] :  92 : 134 : x5C :        : ILLEGAL .
[]] :  93 : 135 : x5D .
[^] :  94 : 136 : x5E .
[_] :  95 : 137 : x5F .
[`] :  96 : 140 : x60 .
[a] :  97 : 141 : x61 .
[b] :  98 : 142 : x62 .
[c] :  99 : 143 : x63 .
[d] : 100 : 144 : x64 .
[e] : 101 : 145 : x65 .
[f] : 102 : 146 : x66 .
[g] : 103 : 147 : x67 .
[h] : 104 : 150 : x68 .
[i] : 105 : 151 : x69 .
[j] : 106 : 152 : x6A .
[k] : 107 : 153 : x6B .
[l] : 108 : 154 : x6C .
[m] : 109 : 155 : x6D .
[n] : 110 : 156 : x6E .
[o] : 111 : 157 : x6F .
[p] : 112 : 160 : x70 .
[q] : 113 : 161 : x71 .
[r] : 114 : 162 : x72 .
[s] : 115 : 163 : x73 .
[t] : 116 : 164 : x74 .
[u] : 117 : 165 : x75 .
[v] : 118 : 166 : x76 .
[w] : 119 : 167 : x77 .
[x] : 120 : 170 : x78 .
[y] : 121 : 171 : x79 .
[z] : 122 : 172 : x7A .
[{] : 123 : 173 : x7B .
[|] : 124 : 174 : x7C :        : ILLEGAL .
[}] : 125 : 175 : x7D .
[~] : 126 : 176 : x7E .
```

CHAPTER

EIGHT

SHELL TOOLS

8.1 dir_to_dot.sh - generate dot(1) graph from directory

dir_to_dot.sh - recursive dot(1) graph from directory tree

usage: dir_to_dot.sh [OPTIONS] [directory]

OPTIONS

option	args	description
-d, –debug		N/I do not execute commands
-h, –help		show this help
–umlx	RX[:TYP]	N/I extract UML diagrams matching RX

|:todo:| document simple database design with || as separator
|:todo:| hidden color set is not properly defined
|:todo:| add –title, –no-title
|:todo:| activity diagram for link processing
|:todo:| add –link-targets, –no-link-targets
|:todo:| add –no-legend, –legend

(see figure 8.1)

(see figure 8.2)

(see figure 8.3)

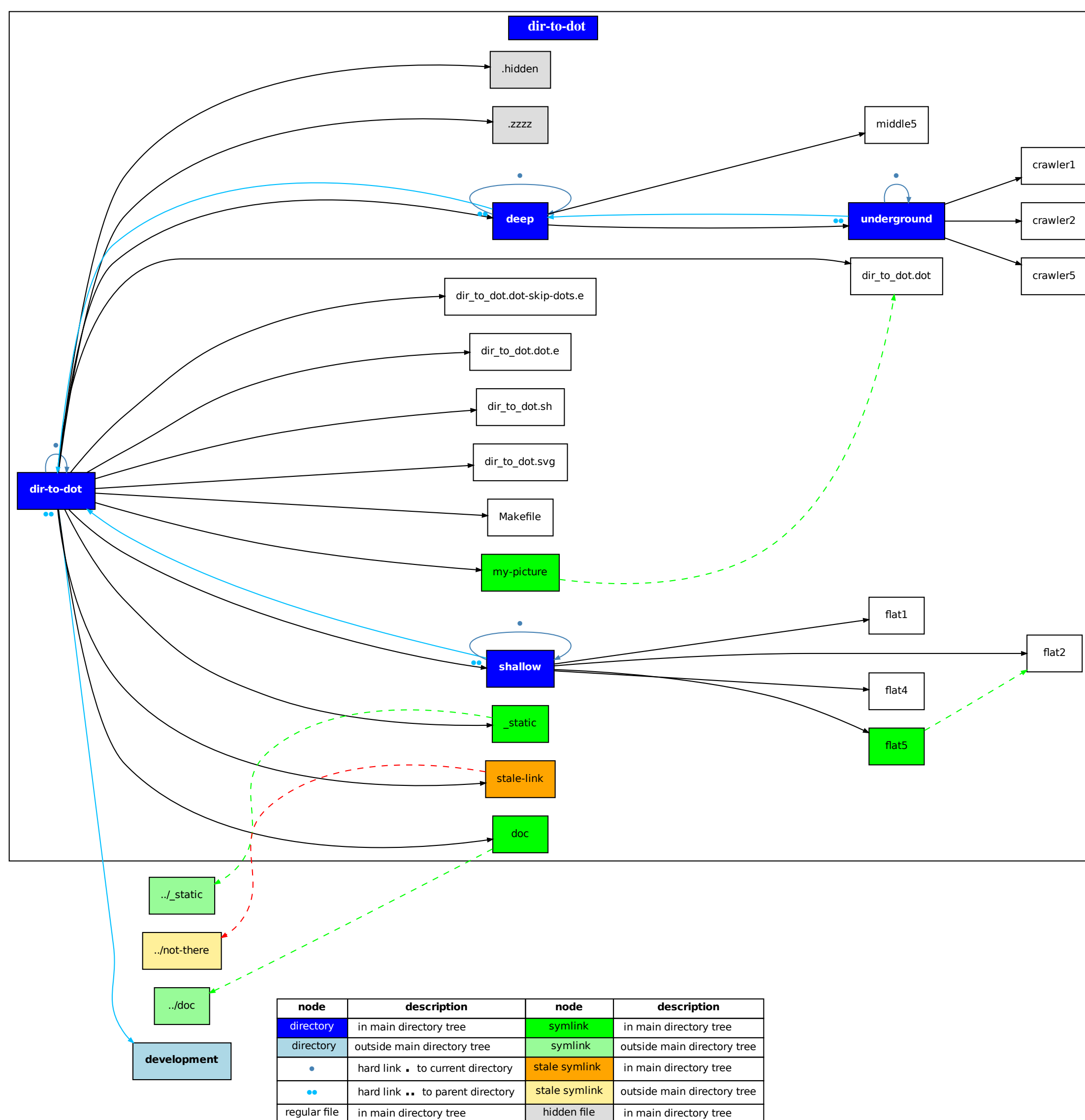


figure 8.1: dir_to_dot.sh generator output

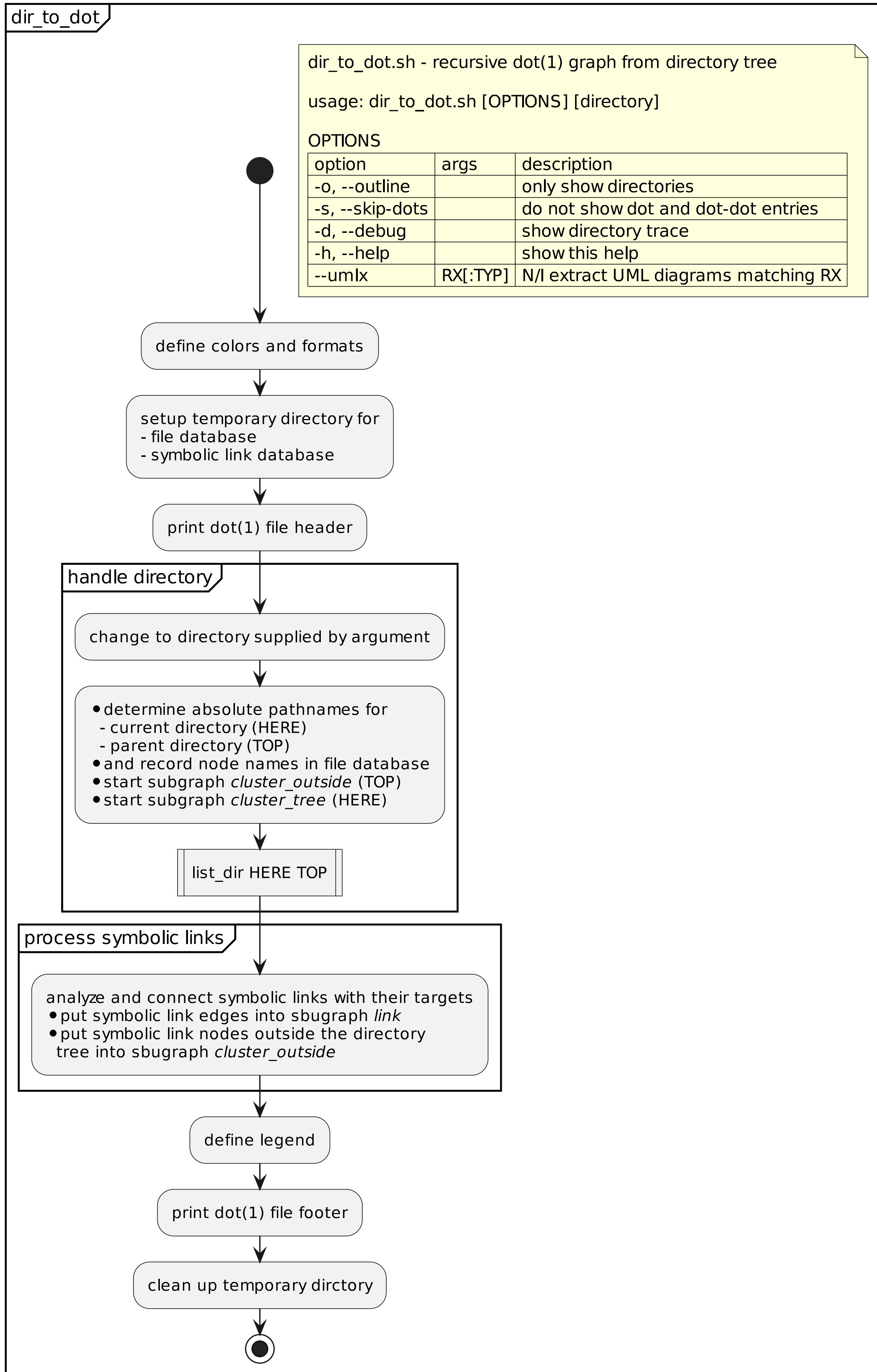


figure 8.2: dir_to_dot.sh generator script

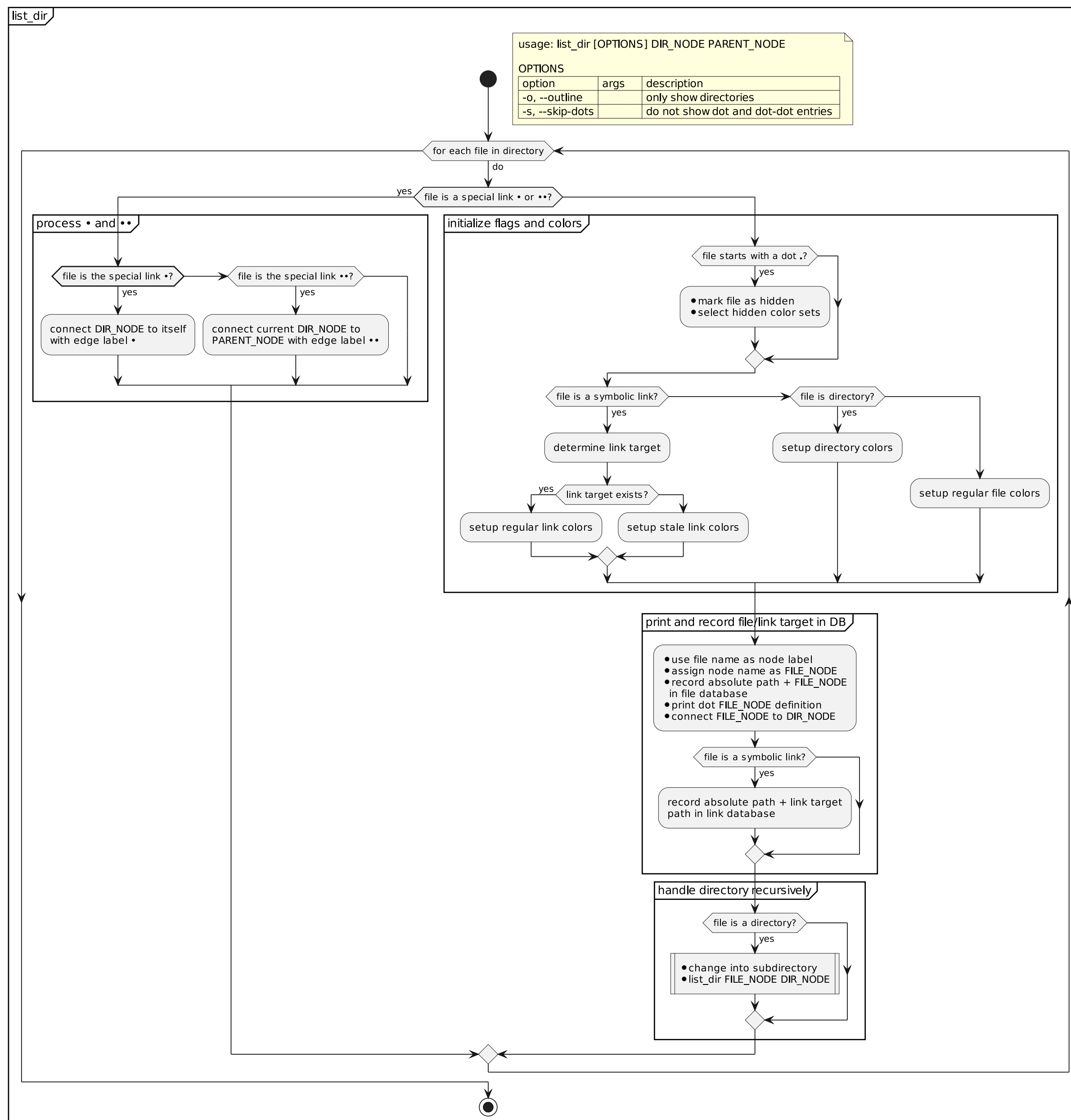


figure 8.3: list_dir recurvise graph generator

CHAPTER NINE

PYTHON

9.1 Language extensions

unpythonic · PyPI

function - JavaScript curry: what are the practical applications? - Stack Overflow

CHAPTER

TEN

COMMON LISP

Links:

- Common Lisp - Wikipedia
- Common Lisp - lisp-lang.org
- Category:Common Lisp implementations - Wikipedia
- Common Lisp HyperSpec (TM)

IDE:

- Portacle - A Portable Common Lisp Development Environment

  features

  - SLIME: The Superior Lisp Interaction Mode for Emacs
  - SBCL - Steel Bank Common Lisp
  - Quicklisp.

  Latest release as of 2019-03-08 22:36:37: https://github.com/portacle/portacle/releases/download/1.3/lin-portacle.tar.xz

  install with:

  ```
  cd
  xz -dc lin-portacle.tar.xz | tar x -
  mkdir bin
  ln -s ../portacle/portacle.run bin/portacle
  ```

  run with:

  ```
  portacle
  ```

Implementations:

- SBCL - Steel Bank Common Lisp (featured by Portacle)
- CLISP - an ANSI Common Lisp Implementation (GNU project), repository: clisp / clisp · GitLab
- GNU Common Lisp - Wikipedia (GNU project)

Libraries:

- Quicklisp loader (ala pip(1))
- CLiki: Current recommended libraries
- Common Lisp libraries overview
- Recommended Libraries | Common Lisp
- GitHub - CodyReichert/awesome-cl: A curated list of awesome Common Lisp frameworks, libraries and other shiny stuff.

GUI:

- CommonQt

CHAPTER

ELEVEN

EVALUATING ALGORITHMS

There are several methods to analyze an algorithm, the following chapters will concern themselves with two possible methods, using the worst-case scenario for InsertionSort. Worst-case InsertionSort takes $\frac{n^2-n}{2}$ comparisons.

Acknowledgments

I'd like to acknowledge the failure that is the german education system, which forces the computer science students in Wuerzburg, Germany to use an idiotic method for analyzing algorithms in their algorithms and data structures class. Or as we say in Germany: "Danke Merkel!"

11.1 Big O notation

See also Big O notation on Wikipedia.

Notation	Name	Description	Formal Definition	Limit Definition
$f(n) = o(g(n))$	Small O; Small Oh	f is dominated by g asymptotically	$\forall k > 0\ \exists n_0\ \forall n > n_0\ \|f(n)\| < k \cdot g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
$f(n) = O(g(n))$	Big O; Big Oh; Big Omicron	$\|f\|$ is bounded above by g (up to constant factor) asymptotically	$\exists k > 0\ \exists n_0\ \forall n > n_0\ \|f(n)\| \leq k \cdot g(n)$	$\limsup_{n\to\infty} \frac{\|f(n)\|}{g(n)} < \infty$
$f(n) = \Theta(g(n))$	Big Theta	f is bounded both above and below by g asymptotically	$\exists k_1 > 0\ \exists k_2 > 0\ \exists n_0\ \forall n > n_0 k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$	$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ (Knuth version)
$f(n) \sim g(n)$	On the order of	f is equal to g asymptotically	$\forall \varepsilon > 0\ \exists n_0\ \forall n > n_0\ \left\|\frac{f(n)}{g(n)} - 1\right\| < \varepsilon$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$
$f(n) = \Omega(g(n))$	Big Omega in number theory (Hardy\u2013Littlewood)	$\|f\|$ is not dominated by g asymptotically	$\exists k > 0\ \forall n_0\ \exists n > n_0\ \|f(n)\| \geq k \cdot g(n)$	$\limsup_{n\to\infty} \left\|\frac{f(n)}{g(n)}\right\| > 0$
$f(n) = \Omega(g(n))$	Big Omega in complexity theory (Knuth)	f is bounded below by g asymptotically	$\exists k > 0\ \exists n_0\ \forall n > n_0\ f(n) \geq k \cdot g(n)$	$\liminf_{n\to\infty} \frac{f(n)}{g(n)} > 0$
$f(n) = \omega(g(n))$	Small Omega	f dominates g asymptotically	$\forall k > 0\ \exists n_0\ \forall n > n_0\ \|f(n)\| > k \cdot \|g(n)\|$	$\lim_{n\to\infty} \left\|\frac{f(n)}{g(n)}\right\| = \infty$

`C-=` runs `rst-adjust()` to extend over-/underlines of titles

11.2 Bakado - the way of the idiot

The way of the idiot is the standard method taught in the course "Algorithmen und Datenstrukturen" at the Julius-Maximilian-University Wuerzburg, as of winter 2018/19. The way of the idiot is named so, because it is tedious and it can take a long time to analyze an algorithm using this method, yet it is universally applicable to analyze any algorithm. Unfortunately it is also the only surefire method to score points on the exam for the "Algorithmen und Datenstrukturen"-course.

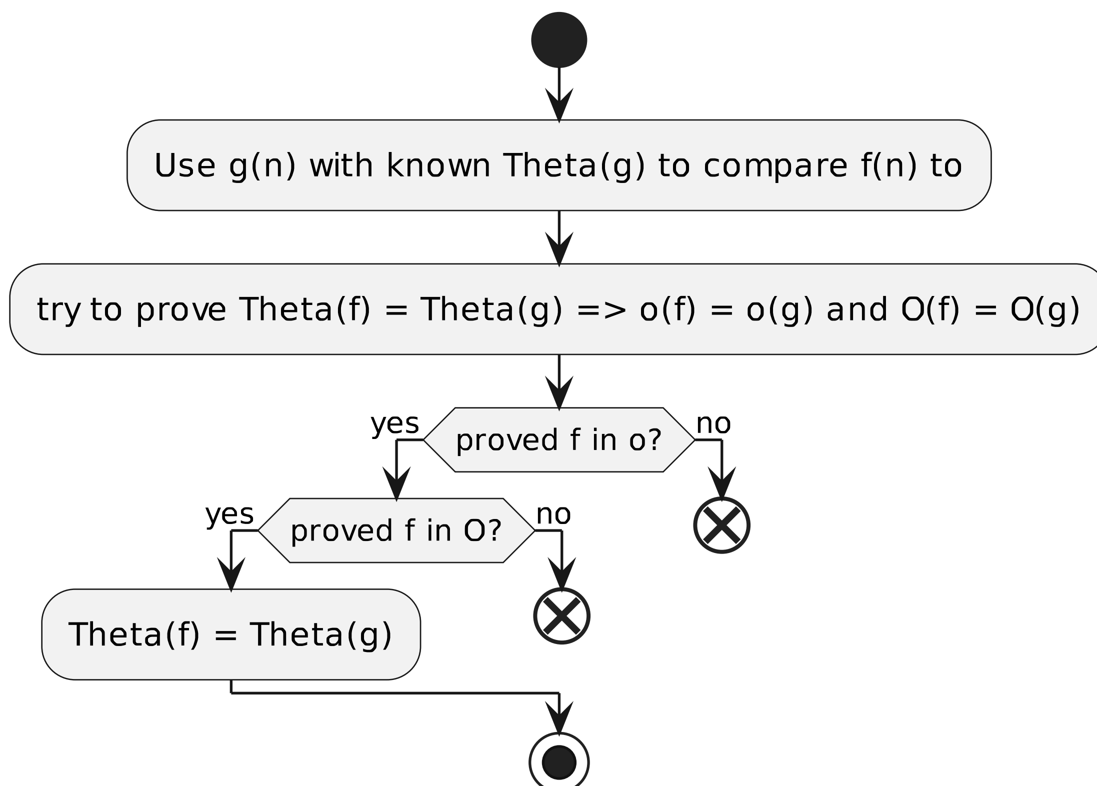
Let $f(n)$ be the worst-case run time of some algorithm f. Let $g(n)$ be the known worst-case run time of some other algorithm g, known to be in runtime class $\Theta(g)$: $g(n) \in \Theta(g)$. Assuming, that $f(n)$ is also in $\Theta(g)$, we now have to prove, that $f(n) \in o(g)$ and $f(n) \in O(g)$. In that case we have found the holy grail of the **real** Θ **housewives**: $\Theta(f) = \Theta(g)$.

$$O(f) = O(g) \not\to \Theta(f) = \Theta(g)$$
$$o(f) = o(g) \not\to \Theta(f) = \Theta(g)$$
$$O(f) = O(g) \wedge o(f) = o(g) \to \Theta(f) = \Theta(g)$$

Let $f(n) = \frac{n^2-n}{2}$ be the worst-case run time of InsertionSort.

We now assume $\Theta(f) = \Theta(g) = \Theta(n^2)$. We now have to prove, that $f(n) \in o(n^2)$ and $f(n) \in O(n^2)$.

The strategy is as follows



11.2.1 Proving $f(n) \in o(g)$

$$o(g) = \{f : \mathbb{N} \to \mathbb{R} | \text{ There exist two constants } c \text{ and } n_0 \text{ such that for all } n \geq n_0 : f(n) \geq c \cdot g(n)\}$$

It is now necessary to find constant values for both c and n_0, such that the definition above holds true, in order to prove that $f(n) \in o(g)$.

In the case of InsertionSort with the worst-case runtime of $\frac{n^2-n}{2}$ we can choose $c = 1$ and $n_0 = 0$, because $n^2 \geq \frac{n^2-n}{2}$ always holds true.

11.2.2 Proving $f(n) \in O(g)$

$$O(g) = \{f : \mathbb{N} \to \mathbb{R} | \text{ There exist two constants } c \text{ and } n_0 \text{ such that for all } n \geq n_0 : f(n) \leq c \cdot g(n)\}$$

It is now necessary to find constant values for both c and n_0, such that the definition above holds true, in order to prove that $f(n) \in O(g)$.

In the case of InsertionSort with the worst-case runtime of $\frac{n^2-n}{2}$ we can choose $c = \frac{1}{4}$ and $n_0 = 2$, because if we pick $c = \frac{1}{4}$ we can solve the following equation for n:

$$\begin{aligned}\frac{1}{4} \cdot n^2 &\leq \frac{n \cdot (n-1)}{2} \\ \frac{1}{4} \cdot n &\leq \frac{n-1}{2} \\ \frac{1}{2} \cdot n &\leq n - 1 \\ n &\leq 2n - 2 \\ 2 &\leq n\end{aligned}$$

this is now our n_0.

11.3 Taidanado - the way of the lazy

The way of the lazy is also known as “Gankona tensaido - the way of the stubborn genius”, the method is known by this name, because it can be applied to analyze any algorithm and it only takes a matter of seconds to apply.

While applying the way of the lazy, one needs to compare the function which one wants to analyze $f(n)$ to another function $g(n)$:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c \to \begin{cases} f(n) \in o(g) & \text{for } c = 0 \\ f(n) \in \Theta(g) & \text{for } 0 < c < \infty \\ f(n) \in O(g) & \text{for } c = \infty \end{cases}$$

In the case of InsertionSort $f(n) = \frac{n^2-n}{2}$ with $g(n) = n^2$: The limit is $\lim_{n \to \infty} \frac{n^2-n}{2} \cdot \frac{1}{n^2} = \frac{1}{2}$, therefore InsertionSort is in $\Theta(n^2)$.

11.4 Satori o aita tensaido - the way of the enlightened genius

Taidanado is extremely useful to get a hint for the constants to choose in the proofs of bakado. And whether to proof or disproof any theorem.

CHAPTER

TWELVE

EXCHANGE VARIABLES WITHOUT TEMPORARY VARIABLE

Was sich gut in Hardware giessen lässt ist noch lange nicht gut für C.

Eine herrliches Schaustück, wie man durch Kontextwechsel und inkorrekte Abstraktion den reinen Blödsinn verzapfen kann und sich auch noch grandios dabei vorkommt.

Zwei Werte vertauschen. Gut und schön, braucht man immer wieder. Die Zwischenvariable ist unästethisch. Naja, Ansichtssache[1].

12.1 Bewerbungsgespräch

Frage Wie vertauscht man in C den Inhalt zweier Variablen ohne Verwendung einer temporären Variablen.

Antwort Nicht rot werden! Unter irgend einem Vorwand das Gebäude verlassen und bitterlich weinend nach einem neuen Arbeitgeber suchen.

Sollte das wegen Kind und Kegel nicht möglich sein, unbedingt darauf achten, dass es dafür keine lebenden Zeugen gibt.

Frage Glaubt denn wirklich irgendjemand, dass das einen Sinn hat?

Antwort Eine Suche auf google.com mit dem Suchausdruck "swap variables with xor" ist da recht aufschlussreich :).

http://www.google.de/search?q=swap+variables+with+xor&hl=de&meta=

Glücklicherweise sind die informierten Stimmen in der Überzahl, aber die uninformierten sind auf der ersten Seite.

12.2 Hardware

Und in der Hardware schaut das auf Signalebene sooo toll aus:

```
X ---+----XOR-->  Q1 (==Y)
      \   /
       XOR
      /   \
Y ---+----XOR-->  Q2 (==X)
```

X	Y	X^Y	Q1 = X^(X^Y)	Q2 = Y^(X^Y)
0	0	0	0	0
0	1	1	1	0
1	0	1	0	1
1	1	0	1	1

[1] Mit Macro gehts auch schöner ... Siehe vswap-macro.c

Yep. Und das ist es auch – in Hardware – Leitung rein / Leitung raus – kein Trick – kein doppelter Boden – Punkt.

Und das war auch schon die eine von zwei echten Lösungen, die tatsächlich keine weitere Hilfestellung benötigt und die in sich geschlossen ist.

Man kann sich auch noch vorstellen, dass sowas Sinn machen kann, um eine Drahtbrücke zu vermeiden (Induktive/Kapazitäre Effekte und Signallaufzeiten in der HF-Magie :)).

Ansonsten würde ja wohl jeder die einfache Lösung vorziehen :)

```
X ---+   +--->  Y
      \ /
       /
      / \
Y ---+   +-->  X
```

Diese hübsche Vermischung von analogen Signalträgern (die sich beliebig aufspalten lassen) und digitaler Verschaltung führt im Softwarebereich zu einem wahrlich grotesken Schauspiel von "Ich auch! Ich auch!"

12.3 Software

Das blödeste Argument, das im Softwarebereich angeführt wird ist die angebliche Schnelligkeit.

12.3.1 Assembler

Die einzig gültige Softwareproblemstellung die mit XOR zu lösen ist, ist gleichzeitig ein API Designfehler, den man aber vielleicht nicht unbedingt vermeiden kann. (Siehe auch lokales Exzerpt von http://www.piclist.com/techref/microchip/math/bit/swap.htm).

Eine Assemblerroutine bekommt zwei Werte in zwei Registern. Dummerweise hat eines der Register Spezialeigenschaften, die benötigt werden, um den zweiten Wert zu behandeln.

Das Szenario sieht so aus:

```
Register  Inhalt        Initialisierung

   R1     WERT          Kann als indirekter Zeiger dienen
   R2     STAPELZEIGER  Kann NICHT als indirekter Zeiger dienen
```

1. Der Wert in R1 muss an die Speicherstelle gelegt werden, auf die R2 zeigt.
2. Einen XCHG-Befehl gibt es nicht.
3. Es dürfen keine anderen Register überschrieben werden.
4. Der Stapel darf nicht verwendet werden.

Hier gibt es nur eine einzige Lösung:

```
xor     R2 |->  R1
xor     R1 |->  R2
xor     R2 |->  R1
mov     R2 |-> (R1)
```

Wenn die Bedingungen gelockert werden, ergibt sich eine Reihe weiterer Varianten.

Wenn ein temporäres Register verwendet werden darf, dann ist die Lösung:

```
mov     R1 |->  R3
mov     R2 |->  R1
mov     R3 |->  R2
mov     R2 |-> (R1)
```

mit Sicherheit genauso schnell, je nach Prozessor sogar schneller.

Lediglich PUSH/POP Varianten, die Speicherzugriffe benötigen, hätten noch eine Chance, als langsamer abgestempelt zu werden:

```
push    R1
mov     R2 |-> R1
pop     R2
mov     R2 |-> (R1)
```

Aber das gilt nur für CPUs ohne Primary Cache :).

Aber am schnellsten ist mit Sicherheit die Lösung, das API anzupassen, sodass die Register richtig belegt sind:

```
mov     R2 |-> (R1)
```

Wenn es darum geht, zwei SPEICHERSTELLEN zu vertauschen, dann wird es so richtig grandios schwachsinnig, selbst wenn der Prozessor Memory-Memory-Operationen kann. Denn Speicherzugriffe sind teuer, wenn kein Cache da ist

XOR-FASSSSSSST (aber praktisch unmöglich, da die meisten CPU's sowas nicht haben :))

```
xor     MEM2 |-> MEM1       ; 2 SP.Zugriffe
xor     MEM1 |-> MEM2       ; 2 SP.Zugriffe
xor     MEM2 |-> MEM1       ; 2 SP.Zugriffe
```

XOR-OPTIMIZED

```
push    R1                  ; 1 SP.Zugriff
push    R2                  ; 1 SP.Zugriff
mov     MEM1 |-> R1         ; 1 SP.Zugriff
mov     MEM2 |-> R2         ; 1 SP.Zugriff
xor     R2   |-> R1
xor     R1   |-> R2
xor     R2   |-> R1
mov     R1   |-> MEM1       ; 1 SP.Zugriff
mov     R2   |-> MEM2       ; 1 SP.Zugriff
pop     R2                  ; 1 SP.Zugriff
pop     R1                  ; 1 SP.Zugriff
```

PLAIN-SWAP

```
push    R1                  ; 1 SP.Zugriff
push    R2                  ; 1 SP.Zugriff
mov     MEM1 |-> R1         ; 1 SP.Zugriff
mov     MEM2 |-> R2         ; 1 SP.Zugriff
mov     R1   |-> MEM2       ; 1 SP.Zugriff
mov     R2   |-> MEM1       ; 1 SP.Zugriff
pop     R2                  ; 1 SP.Zugriff
pop     R1                  ; 1 SP.Zugriff
```

Selbst Intel-CPU's haben zwei Register, die vom Compiler als Scratch behandelt werden, d.h. sie werden üblicherweise nicht gesichert ax, dx).

Damit ergibt sich als optimalste Lösung

PLAIN-SWAP

```
mov     MEM1 |-> R1         ; 1 SP.Zugriff
mov     MEM2 |-> R2         ; 1 SP.Zugriff
mov     R1   |-> MEM2       ; 1 SP.Zugriff
mov     R2   |-> MEM1       ; 1 SP.Zugriff
```

12.3.2 Interpreter

Und die Visual Basic Dumpfbacken müssen natürlich auch plärren (siehe lokale Kopie von dead link http://www.pinnaclepublishing.com/VB/VBmag.nsf/0/15D2712EB6378291852568D8006AFFEE):

> "Here's a quick tip. I think the following code provides a good alternative that's both faster and uses less memory than the usual method of declaring a temporary variable and using that to swap the variables.
>
> ```
> Public Sub SwapLong(ByRef lngFirst As Long, ByRef lngSecond As Long)
>    lngFirst = lngFirst Xor lngSecond
>    lngSecond= lngFirst Xor lngSecond
>    lngFirst = lngFirst Xor lngSecond
> End Sub"
> ```

Die Alternative:

```
Public Sub SwapLong(ByRef lngFirst As Long, ByRef lngSecond As Long)
   Temp = lngFirst
   lngFirst = lngSecond
   lngSecond = Temp
End Sub
```

Also, wie macht's denn der Interpreter so im allgemeinen?

Zuweisung:: Variable suchen Expression auswerten Wert zuweisen

Expression == Variable:: Variable suchen Wert auslesen

Expression == (E1 Xor E2):: E1 auswerten E2 auswerten Werte verknüpfen (das temporär angelegte) Ergebnis liefern

XOR Lösung

```
Variable lngFirst suchen
Variable lngFirst suchen
Wert auslesen
Variable lngSecond suchen
Wert auslesen
Werte verknüpfen (und hier haben wir doch eine temporäre Variable :))
Wert zuweisen

Variable lngSecond suchen
Variable lngFirst suchen
Wert auslesen
Variable lngSecond suchen
Wert auslesen
Werte verknüpfen (und hier haben wir doch eine temporäre Variable :))
Wert zuweisen

Variable lngFirst suchen
Variable lngFirst suchen
Wert auslesen
Variable lngSecond suchen
Wert auslesen
Werte verknüpfen (und hier haben wir doch eine temporäre Variable :))
Wert zuweisen
```

NORMALE LÖSUNG

```
Temp anlegen (Und das soll jetzt sooo lange brauchen und sooo viel
              Speicher fressen ?:))
Variable lngFirst suchen
Wert auslesen
Wert zuweisen

Variable lngFirst suchen
Variable lngSecond suchen
Wert auslesen
Wert zuweisen
```

(continues on next page)

(continued from previous page)

```
Variable lngSecond suchen
Variable Temp suchen
Wert auslesen
Wert zuweisen
```

Das ist jetzt schon lächerlich, oder nicht? :)

12.3.3 Compiler

Tja, und in C :)))

Bei Speicherzugriffen gilt das gleiche wie für Assembler.

Also tauschen wir mal nur zwei Variablen aus, die weder aus dem Speicher kommen, noch in den Speicher gehen …

```
{
    int erste  = 55aa;
    int zweite = ff00;

    result = erste * zweite;

    /* Und jetzt der grosse Austausch! */

    result = zweite * erste;
}
```

12.3.4 Details

Ja, ja das allseits beliebte XOR …

a	b	a' == (a xor b)	b' == a' xor b	a' xor b'
1	1	0	1	1
1	0	1	1	0
0	1	1	0	1
0	0	0	0	0

Fein.

```
a' = a  xor b // a enthält nun a'
b' = a' xor b
a  = a' xor b'
```

Oder:

```
a ^= b;
b ^= a;
a ^= b;
```

Na, wenn wir schon dabei sind …

```
a ^= ( b ^= ( a ^= b ));
```

So. Jetzt kann's mit Sicherheit keiner mehr lesen, aber wenn's in einer Zeile steht, dann muss es wohl schneller sein als der Standard-Dreizeiler ….

```
c = a;
a = b;
b = c;
```

Also, für so einen hochoptimierten Algrithmus lohnt sich mit Sicherheit ein wenig Inline-Assembler. Portabilität muss ja nicht immer gleich die höchste Priorität haben . . .

```
Intel:                      PPC:
        movl a,%eax                 lwz 0,a@l(29)
        movl b,%edx                 lwz 9,b@l(28)
        xorl %edx,%eax              xor 0,0,9
        xorl %eax,%edx              xor 9,9,0
        xorl %edx,%eax              xor 0,0,9
        movl %eax,a                 stw 0,a@l(29)
        movl %edx,b                 stw 9,b@l(28)
```

Da können wir aber noch ein wenig feilen . . .

```
Intel:                      PPC:
        movl a,%eax                 lwz 0,a@l(29)
        movl b,%edx                 lwz 9,b@l(28)
        movl %edx,a                 stw 9,a@l(29)
        movl %eax,b                 stw 0,b@l(28)
```

Oder bekommt's der Kompiler besser hin? (Siehe vswap.c).

Also schau 'mer mal (mit volatile und verschiedenen Optimierungen):

```
XOR1 -O0:                   XOR2 -O0:                   SWAP -O0:
        movl a,%eax                 movl a,%eax                 movl a,%eax
        movl b,%edx                 movl b,%edx                 movl %eax,-4(%ebp)
        xorl %edx,%eax              xorl %edx,%eax              movl b,%eax
        movl %eax,a                 movl %eax,a                 movl %eax,a
        movl a,%eax                 movl b,%eax                 movl -4(%ebp),%eax
        movl b,%edx                 movl a,%edx                 movl %eax,b
        xorl %edx,%eax              xorl %edx,%eax
        movl %eax,b                 movl %eax,b
        movl b,%eax                 movl a,%eax
        movl a,%edx                 movl b,%edx
        xorl %edx,%eax              xorl %edx,%eax
        movl %eax,a                 movl %eax,a
```

```
XOR1 -Ox:                   XOR2 -Ox:                   SWAP -Ox:
        movl a,%eax                 movl a,%eax                 movl a,%edx
        movl b,%edx                 movl b,%edx                 movl b,%eax
        xorl %edx,%eax              xorl %edx,%eax              movl %eax,a
        movl %eax,a                 movl %eax,a                 movl %edx,b
        movl a,%edx                 movl b,%eax
        movl b,%eax                 movl a,%edx
        xorl %edx,%eax              xorl %edx,%eax
        movl %eax,b                 movl %eax,b
        movl b,%edx                 movl a,%eax
        movl a,%eax                 movl b,%edx
        xorl %edx,%eax              xorl %edx,%eax
        movl %eax,a                 movl %eax,a
```

Na so ein Mist :(. Wenn das Pointer wären und sich wirklich eins der Dinger zwischendrin ändert, dann geht das mit dem XOR voll daneben. Der Compiler ist ja wirklich doof.

Hmm, dann halt ohne volatile . . . (Kann man ja auch weg-casten! Dann weiss auch gleich jeder, dass wir eigentlich nur tauschen wollen. Und dass wir jetzt ZWEI temporäre Variablen brauchen soll uns auch nicht weiter stören, weil der Algorithmus ja sooo schöööön ist :))

```
XOR1 -O0:                   XOR2 -O0:                   SWAP -O0:
        movl a,%eax                 movl b,%eax                 movl a,%eax
        xorl b,%eax                 xorl %eax,a                 movl %eax,-4(%ebp)
        movl %eax,%edx              movl a,%eax                 movl b,%eax
        movl %edx,a                 xorl %eax,b                 movl %eax,a
        movl %edx,%eax              movl b,%eax                 movl -4(%ebp),%eax
        xorl b,%eax                 xorl %eax,a                 movl %eax,b
        movl %eax,%edx
        movl %edx,b
        xorl %edx,a
```

Schön! mit -O0 haben wir aber jetzt einen gewaltigen Vorteil! (Na gut, man muss auch noch wissen, dass es der Compiler schön einfach notiert haben will.)

```
XOR1 -Ox:                XOR2 -Ox:                SWAP -Ox:
        movl a,%edx              movl a,%edx              movl a,%eax
        xorl b,%edx              xorl b,%edx              movl b,%edx
        movl a,%eax              movl a,%eax              movl %edx,a
        movl %eax,b              movl %eax,b              movl %eax,b
        xorl %eax,%edx           xorl %eax,%edx
        movl %edx,a              movl %edx,a
```

Und mit Optimierung schon wieder nicht, weil der blöde Compiler nicht kapiert, dass wir nur zwei Variablen tauschen wollen :(. Wenn er doch schon merkt, dass das eine XOR unnötig ist – warum merkt er denn das zweite nicht?

Herr im Himmel! Das kann doch nur an dem blöden Intel-Prozessor liegen, weil der keine Memory-Memory-Operationen kann.

Wie sieht's denn da mit dem PPC aus? (Die volatile-Version wollen wir glaub' ich nicht sehen . . .)

```
XOR1 -Ox:                XOR2 -Ox:                SWAP -Ox:
        lis 9,a@ha               lis 9,a@ha               lis 9,a@ha
        lis 11,a@ha              lis 11,a@ha              lwz 0,a@l(9)
        lis 10,b@ha              lis 10,b@ha              stw 0,16(31)
        lis 8,b@ha               lwz 0,a@l(11)            lis 9,a@ha
        lis 7,a@ha               lwz 11,b@l(10)           lis 11,b@ha
        lis 6,a@ha               xor 0,0,11               lwz 0,b@l(11)
        lis 5,b@ha               stw 0,a@l(9)             stw 0,a@l(9)
        lwz 0,a@l(6)             lis 9,b@ha               lis 9,b@ha
        lwz 5,b@l(5)             lis 11,b@ha              lwz 0,16(31)
        xor 6,0,5                lis 10,a@ha              stw 0,b@l(9)
        mr 0,6                   lwz 0,b@l(11)
        stw 0,a@l(7)             lwz 11,a@l(10)
        lwz 7,b@l(8)             xor 0,0,11
        xor 8,0,7                stw 0,b@l(9)
        mr 0,8                   lis 9,a@ha
        stw 0,b@l(10)            lis 11,a@ha
        lwz 11,a@l(11)           lis 10,b@ha
        xor 0,11,0               lwz 0,a@l(11)
        stw 0,a@l(9)             lwz 11,b@l(10)
                                 xor 0,0,11
                                 stw 0,a@l(9)
```

Ach verdammt. Jetzt ist der -O0 Vorteil vom Intel-Prozessor hin :(. Na macht nix, wir benutzen halt einen #define auf den Prozessor, um die jeweils optimalere Variante zu verwenden :).

```
XOR1 -Ox:                XOR2 -Ox:                SWAP -Ox:
        lwz 9,b@l(28)            lwz 9,b@l(28)            lwz 6,a@l(29)
        lwz 0,a@l(29)            lwz 0,a@l(29)            lwz 5,b@l(28)
        xor 0,0,9                xor 0,0,9                stw 5,a@l(29)
        xor 9,9,0                xor 9,9,0                stw 6,b@l(28)
        xor 0,0,9                xor 0,0,9
        stw 9,b@l(28)            stw 9,b@l(28)
        stw 0,a@l(29)            stw 0,a@l(29)
```

Hmm. Der PPC hat offensichtlich auch keine Memory-Memory-Operationen :(. (Einen HPPA hätt ich noch anzubieten. Aber der ist auch ein RISC :)).

Also wenn ich mir das so anschaue, glaube ich, dass die Compiler- Schreiber noch ordentlich ran müssen, bevor sie behaupten können, sie wären in der Lage C-Code fast wie mit der Hand optimieren zu können :).

PS: So geht's übrigens auch . . .

```
b = ( a ^ b ) ^ ( a = b );
```

. . . jedenfalls meistens **:)** [2].

[2] Zumindest kommt der beste XOR Assembler dabei raus (4 Speicherzugriffe). Und beim PPC merkt der Compiler sogar endlich, was gespielt wird!

Nur der saudumme ANSI-Standard sagt, dass der Compiler-Hersteller bei kommutativen Operationen die Reihenfolge der Auswertung ändern darf. Und wenn er das wirklich mal macht (SGI Compiler mit -n32!), dann wird's im Speicher wenigstens schön einheitlich:

```
b = ( a = b ) ^ ( a ^ b );
```

PPS: Die Ausgabe des Programmchens . . .

```
--init--                                                              a: 0x55aa55aa b: 0xff00ff00
x: a ^= ( b^= ( a ^= b ));                                       res: a: 0xff00ff00 b: 0x55aa55aa
x: ((int) a ) ^= (((int) b )^= (((int) a ) ^= ((int) b ))); res: a: 0x55aa55aa b: 0xff00ff00
x: c ^= ( d^= ( c ^= d ));                                       res: a: 0xff00ff00 b: 0x55aa55aa
x: a ^= b; b ^= a; a ^= b;                                       res: a: 0x55aa55aa b: 0xff00ff00
x: c = a; a = b; b = c;                                          res: a: 0xff00ff00 b: 0x55aa55aa
x: b = ( a ^ b ) ^ ( a = b );                                    res: a: 0x55aa55aa b: 0xff00ff00
x: b = ( a = b ) ^ ( a ^ b );                                    res: a: 0xff00ff00 b: 0xff00ff00
```

PPPS:

Und den Schwachsinn kann man auch noch des Langen und des Breiten im Internet nachlesen :))))))))))))) (Siehe lokale Kopie von dead link http://www.con.wesleyan.edu/~eric/cpp-tips/xor-swap.html).

Vor allen Dingen:

> "Using this as an inline function, where A and B are references to integers, increased my quicksort implementation by a lot (several tenths)"

```
Intel:                     PPC:
        movl b,%edx                lwz 6,a@l(26)
        movl %edx,%eax             lwz 5,b@l(27)
        xorl a,%eax                stw 6,b@l(27)
        movl %edx,a                stw 5,a@l(26)
        xorl %edx,%eax
        movl %eax,b
```

CHAPTER

THIRTEEN

MOV IS TURING-COMPLETE

No, **mov** is **not** Turing-complete. There must be

- either an additional **jmp** instruction
- or separate data and program memory, where program memory is
  - finite with wrap-around,
  - or infinite.

See also rebuttal on reddit mov is Turing-complete by Stephen Dolan: ReverseEngineering.

1. Eine Turing-Maschine mit ausschließlich **mov**-Instruktion ist **nicht realisierbar**.

   Das großspurig angekündigte mov is Turing-complete - Cambridge Computer Laboratory (Dokument ist nicht mehr verfügbar. Lokaler Link) bringt nicht neues und auf Seite 4 wird die Katze dann aus dem Sack gelassen. Es braucht einen **jmp** Befehl:

   > . . . in a suitable state to run the program again, so we use our single unconditional jump to loop back to the start:
   >
   > ```
   > jmp start
   > ```
   >
   > This simulates an arbitrary Turing machine, using only mov (**and a single jmp** to loop the program).

2. Es gibt auch keinen C-Compiler, der nur mit **mov** geschrieben ist (obwohl man das natürlich realisieren kann, wenn man sehr viel Speicher übrig hat).

   Es gibt eine LCC-Erweiterung M/o/Vfuscator2 die ein C-Programm in **mov**-Instruktionen übersetzt. Dieses Programm selbst ist aber in C geschrieben.

   Und der Autor redet sich ebenfalls die Welt schön. Im Abschnitt *Notes*:

   > While Dolan's paper required a jmp instruction, the M/o/Vfuscator does not - it uses a **faulting mov instruction** to achieve the infinite execution loop.
   >
   > If you're worried that this is still "jumping", the same effect could be achieved through
   >
   > a) **pages aliased to the same address**,
   >
   > b) **wrapping execution around the upper range of memory**,
   >
   > c) **ring 0 exception handling**,
   >
   > d) or simply **repeating the mov loop indefinitely**.
   >
   > . . .

   Die Punkte a) - c) sind alles mehr oder weniger verdeckte Sprünge!

   Wobei Punkt a) wenigstens noch als Endlosband denkbar ist. Das sollte man aber auch klar und deutlich spezifizieren. Auf jeden Fall ist es dann Essig mit gemeinsamem Programm- und Datenspeicher. Die müssen dann auf jeden Fall getrennt sein. Und damit braucht es eine nichtexistente (aber realisierbare) CPU.

Punkt d) ist illusorischer Quatsch. In diesem Fall haben wir eine CPU mit getrenntem Programmspeicher und Datenspeicher, wobei der Programmspeicher unendlich groß ist. Also nichts realisierbares, weil es keine finiten Register geben kann, die eine unendlich große Adresse speichern könnten. Mit einem $\aleph_0$ basierten Zahlensystem bräuchte so ein Register allerdings nur ein A-Bit. Dagegen wären wohl Q-Bits bloß ein schwacher Abklatsch.

3. Wenn man also sowieso einen **jmp** braucht, dann kann man auch gleich einen bedingten Sprung nehmen und hat damit im Wesentlichen genau das existierende und realisierte Konzept einer von-Neumann-Maschine. Die zusätzlichen arithmetischen, logischen und Schiebeoperationen sind dann halt einfach sinnvoller Luxus.

Das ganze ist also ein reichlich sinnloses Unterfangen, das nicht mal intellektuell anspruchsvoll ist, weil es mit Desinformation arbeitet. Irgenwelche Schlußfolgerungen daraus zu ziehen ist hanebüchen ;-) Brainfuck und Malbolge sind wenigstens amüsant und nicht so prätentiös.

CHAPTER

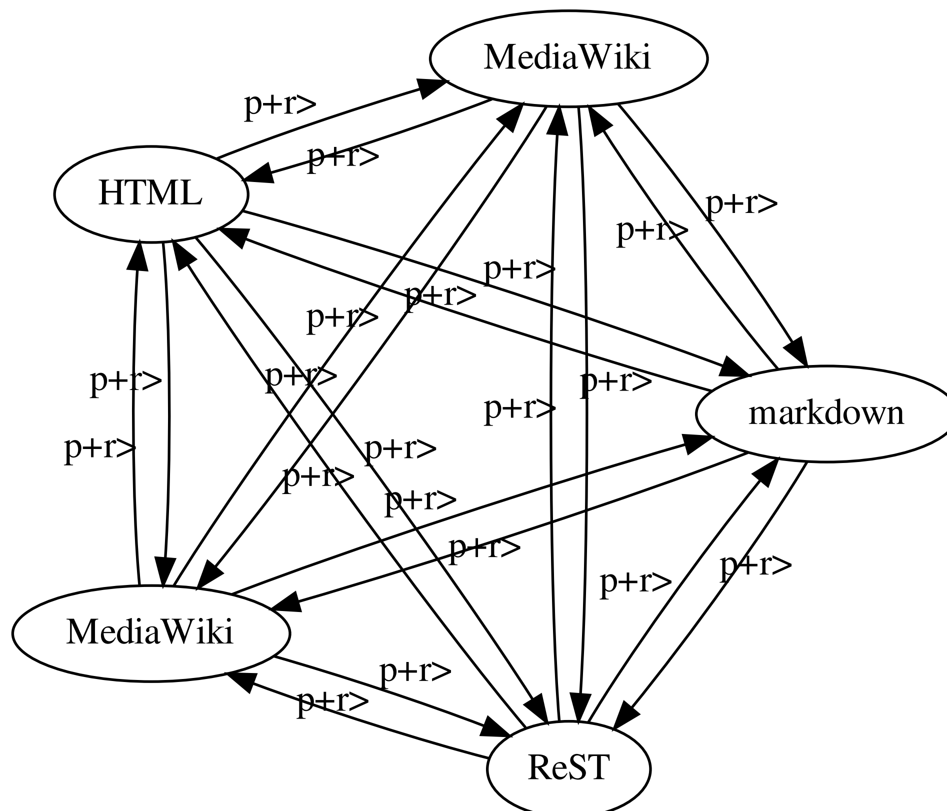
FOURTEEN

MARKUP CONVERTER DESIGN

pandoc(1) most probably implements a central generic DOM for markup conversion vs. a direct conversion between markup languages. The difference being development requirements of $\Theta(n)$ vs. $\Theta(n^2)$.

14.1 Direct conversion between markup languages

For each new markup language n, a parser supporting $n - 1$ renderers must be written for translating the new language to all implemented languages. For $n-1$ parsers a renderer must be added for translating the implemented languages to the new language.



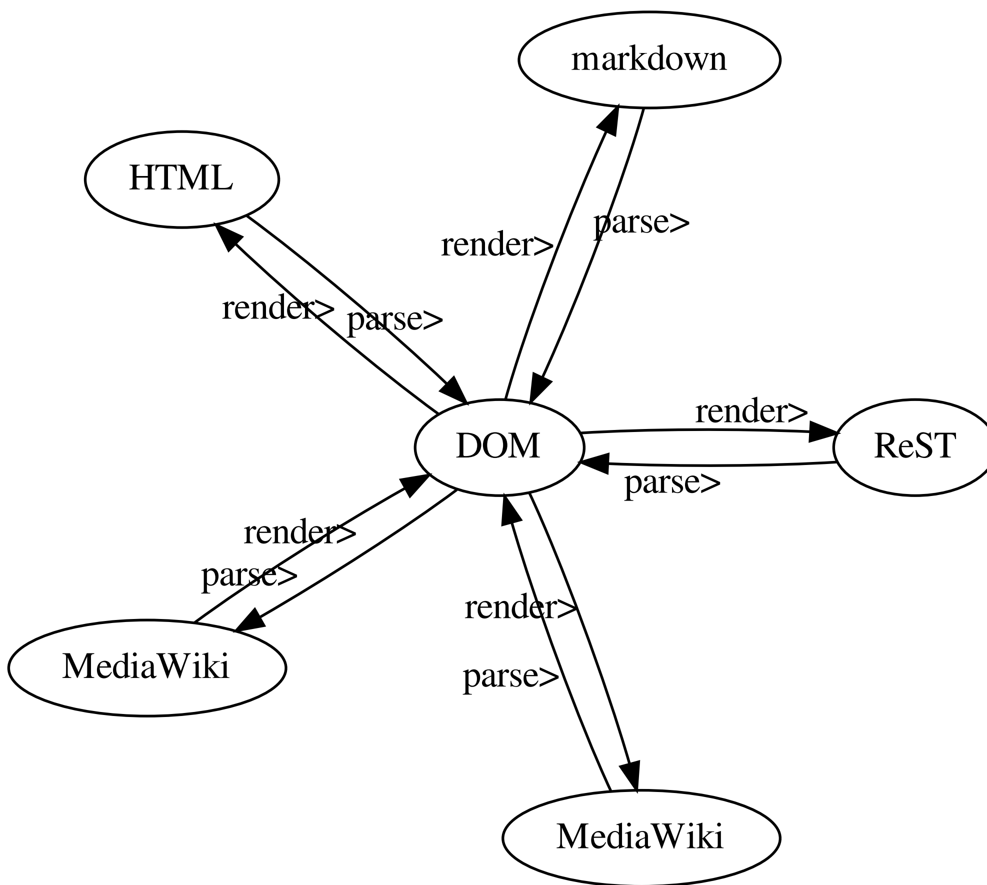
Progression for number of nodes and total number of edges / 2:

nodes	0	1	2	3	4	5	6	7	8	9	10
edges/2	0	0	1	3	6	10	15	21	28	36	45

Identified on OEIS as sequence A161680 $a(n) = \binom{n}{2} = \frac{n^2-n}{2}$ (zero followed by a000217, triangular numbers: $a(n) = \binom{n+1}{2} = \frac{n(n+1)}{2}$).

14.2 Conversion via central abstract DOM

For each new markup language one parser (markup -> DOM) and one renderer (DOM -> markup) must be written to fully support translations between all previously implemented languages and the new language.



Progression for number of nodes and total number of edges / 2:

nodes	0	1	2	3	4	5	6	7	8	9	10
edges/2	0	0	1	2	3	4	5	6	7	8	9

$a(n) = \max(0, n-1)$ (see also A289207).

CHAPTER FIFTEEN

SHELL PIDS

See also linux - How to get PID from forked child process in shell script - Stack Overflow.

The script, child and parent PIDs can always be determined correctly in the parent script independent of the method employed to start it:

```
child () { :; }

child &
( child ) &
sh -c "child () { :; }; child" &

script_pid="${$}"     # PID of the running shell script
child_pid="${!}"      # last started child process
parent_pid="${PPID}" # PID of parent process
```

Whether the information is accesible in the child process depends on how the child process has been started.

With the function:

```
child ()
{
    sleep "$( expr "${2-1}" '*' 5 )"
    script_pid="${3-}"
    child_pid="${$}"
    parent_pid="${PPID}"
    message="CHILD ${1-}"
    test x"${script_pid}" != x"${child_pid}" || message="${message}"' UNFORTUNATELY WRONG!'
    report
    sleep 20
}
```

and the asynchrounous calls:

```
child FUNC "${indx}" "${$}" &

( child SUB "${indx}" "${$}" ) &

sh -c "${FUNCTIONS}
child SH "${indx}" \"${$}\"" &
```

the output shows, that only the function invoked with an explicit `sh -c` has access to the correct PID/PPID information:

```
# --------------------------------------------------
#   :PRC:   CHILD FUNC UNFORTUNATELY WRONG!
#   :DBG:   script_pid      : [20634]
#   :DBG:   child_pid       : [20634]
#   :DBG:   parent_pid      : [10857]
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
0  1114 10857 10853  20   0 2756840 1942820 poll_s Ssl ?       95:15 emacs
0  1114 20634 10857  20   0   4636  1732 wait   Ss+  pts/22     0:00 sh /tmp/check_child_and_
↪parent_pids.sh

# --------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
#    :PRC:    CHILD SUB UNFORTUNATELY WRONG!
#    :DBG:    script_pid      : [20634]
#    :DBG:    child_pid       : [20634]
#    :DBG:    parent_pid      : [10857]
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
0  1114 10857 10853  20   0 2756840 1942820 poll_s Ssl ?       95:15 emacs
0  1114 20634 10857  20   0   4636  1732 wait   Ss+  pts/22     0:00 sh /tmp/check_child_and_
↪parent_pids.sh

# --------------------------------------------------
#    :PRC:    CHILD SH
#    :DBG:    script_pid      : [20634]
#    :DBG:    child_pid       : [20658]
#    :DBG:    parent_pid      : [20634]
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
0  1114 20634 10857  20   0   4636  1732 wait   Ss+  pts/22     0:00 sh /tmp/check_child_and_
↪parent_pids.sh
0  1114 20658 20634  20   0   4636  1720 wait   S+   pts/22     0:00 sh -c show_processes ()
↪{     test -z
```

The complete test script is:

```
FUNCTIONS="$(
cat <<'EOF'
show_processes ()
{
    test -z "${1-}" || printf "%s\n" "${1}"

    _sep=
    pids_rx=
    for _pid in "${script_pid}" "${child_pid}" "${parent_pid}"
    do
        test -n "${_pid}" || continue
        pids_rx="${pids_rx}${_sep}${_pid}"
        _sep='\|'
    done

    ps axl | head -1
    ps axl | grep '^[^ ]*  *[^ ]*  *\('"${pids_rx}"'\)' | cut -c -107 | sort -k 3
}

report ()
{
    printf "%s\n" ""
    printf >&2 "# --------------------------------------------------\n"
    printf >&2 "#   "":PRC:   %s\n" "${message}"
    test -z "${script_pid}" || \
    printf >&2 "#   "":DBG:   %-${dbg_fwid-15}s: [%s]\n" "script_pid" "${script_pid}"
    test -z "${child_pid}" || \
    printf >&2 "#   "":DBG:   %-${dbg_fwid-15}s: [%s]\n" "child_pid" "${child_pid}"
    test -z "${parent_pid}" || \
    printf >&2 "#   "":DBG:   %-${dbg_fwid-15}s: [%s]\n" "parent_pid" "${parent_pid}"
    show_processes
}

child ()
{
    sleep "$( expr "${2-1}" '*' 5 )"
    script_pid="${3-}"
    child_pid="${$}"
    parent_pid="${PPID}"
    message="CHILD ${1-}"
    test x"${script_pid}" != x"${child_pid}" || message="${message}"' UNFORTUNATELY WRONG!'
    report
    sleep 20
}
EOF
)"

eval "${FUNCTIONS}"
```

(continues on next page)

(continued from previous page)

```
shell_report ()
{
    script_pid="${$}"
    child_pid="${!}"
    parent_pid="${PPID}"
    sleep 1
    message="SHELL ${indx}"
    report
    indx="$( expr "${indx}" + 1 )"
}

indx=1

child FUNC "${indx}" "${$}" &
shell_report

( child SUB "${indx}" "${$}" ) &
shell_report

sh -c "${FUNCTIONS}
child SH "${indx}" \"${$}\"" &
shell_report

sleep "$( expr "${indx}" '*' 5 )"
```

The complete output is:

```
# --------------------------------------------------
#   :PRC:   SHELL 1
#   :DBG:   script_pid      : [20634]
#   :DBG:   child_pid       : [20636]
#   :DBG:   parent_pid      : [10857]
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
0  1114 10857 10853  20   0 2756840 1942820 poll_s Ssl ?       95:15 emacs
0  1114 20634 10857  20   0   4636  1728 wait   Ss+  pts/22     0:00 sh /tmp/check_child_and_
↪parent_pids.sh
1  1114 20636 20634  20   0   4636   104 wait   S+   pts/22     0:00 sh /tmp/check_child_and_
↪parent_pids.sh

# --------------------------------------------------
#   :PRC:   SHELL 2
#   :DBG:   script_pid      : [20634]
#   :DBG:   child_pid       : [20647]
#   :DBG:   parent_pid      : [10857]
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
0  1114 10857 10853  20   0 2756840 1942820 -   Rsl  ?         95:15 emacs
0  1114 20634 10857  20   0   4636  1732 wait   Ss+  pts/22     0:00 sh /tmp/check_child_and_
↪parent_pids.sh
1  1114 20647 20634  20   0   4636   108 wait   S+   pts/22     0:00 sh /tmp/check_child_and_
↪parent_pids.sh

# --------------------------------------------------
#   :PRC:   SHELL 3
#   :DBG:   script_pid      : [20634]
#   :DBG:   child_pid       : [20658]
#   :DBG:   parent_pid      : [10857]
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
0  1114 10857 10853  20   0 2756840 1942820 poll_s Ssl ?       95:15 emacs
0  1114 20634 10857  20   0   4636  1732 wait   Ss+  pts/22     0:00 sh /tmp/check_child_and_
↪parent_pids.sh
0  1114 20658 20634  20   0   4636   928 wait   S+   pts/22     0:00 sh -c show_processes ()
↪{     test -z

# --------------------------------------------------
#   :PRC:   CHILD FUNC UNFORTUNATELY WRONG!
#   :DBG:   script_pid      : [20634]
#   :DBG:   child_pid       : [20634]
#   :DBG:   parent_pid      : [10857]
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
```

(continues on next page)

(continued from previous page)

```
0  1114 10857 10853  20   0 2756840 1942820 poll_s Ssl ?       95:15 emacs
0  1114 20634 10857  20   0   4636  1732 wait   Ss+  pts/22     0:00 sh /tmp/check_child_and_
↪parent_pids.sh

# --------------------------------------------------
#   :PRC:   CHILD SUB UNFORTUNATELY WRONG!
#   :DBG:   script_pid     : [20634]
#   :DBG:   child_pid      : [20634]
#   :DBG:   parent_pid     : [10857]
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
0  1114 10857 10853  20   0 2756840 1942820 poll_s Ssl ?       95:15 emacs
0  1114 20634 10857  20   0   4636  1732 wait   Ss+  pts/22     0:00 sh /tmp/check_child_and_
↪parent_pids.sh

# --------------------------------------------------
#   :PRC:   CHILD SH
#   :DBG:   script_pid     : [20634]
#   :DBG:   child_pid      : [20658]
#   :DBG:   parent_pid     : [20634]
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
0  1114 20634 10857  20   0   4636  1732 wait   Ss+  pts/22     0:00 sh /tmp/check_child_and_
↪parent_pids.sh
0  1114 20658 20634  20   0   4636  1720 wait   S+   pts/22     0:00 sh -c show_processes ()
↪{     test -z
```

CHAPTER SIXTEEN

EMACS EXTENSION

16.1 Emacs-Erweiterungen mit Cut-Paste-Modify

Es ist unsinnig - wie immer - mit einer nackten Funktion anzufangen. Das Stichwort ist hier *Cut-Paste-Modify*. Immer Beispiele kopieren und abändern! So lange, bis der Groschen gefallen ist. Alles andere ist *Schule* und blanker Unsinn.

Man fängt in den wenigsten Fällen mit einer Funktion oder gar einem Package an. Die Übersicht geht dabei komplett verloren und das Ergebnis ist mager.

Was soll die Funktion also können?

> Wenn ich [die Funktion] ausführe, dann wird ein neuer Shell-Buffer geöffnet, der als Namen aber standardmäßig “screenlog-<heutiges-datum>” heißen soll (ggf. mit Präfix ein anderer Name.

Darum …

16.1.1 Erst mal Proof-of-Concept

Wenn man alle notwendigen Elemente per Makro zusammenstellen kann, dann hat man von Anfang an das richtige Konzept und automatisch auch die richtigen Befehle, die man dann in einerm interaktiven Kommando benötigt.

Wie man einen Shell-Buffer bekommt, weißt du ja schon. Es geht mit `M-x shell RET`. Damit ist auch klar, dass man die Funktion `shell` benötigt.

Erst mal sehen, was die Funktion so anbietet. Mit `C-h f shell RET` wirst du dabei fündig (siehe figure 16.1).

```
shell is an interactive autoloaded compiled Lisp function in
‘shell.el’.

(shell &optional BUFFER)

Run an inferior shell, with I/O through BUFFER (which defaults to ‘*shell*’).
Interactively, a prefix arg means to prompt for BUFFER.
If ‘default-directory’ is a remote file name, it is also prompted
to change if called with a prefix arg.

If BUFFER exists but shell process is not running, make new shell.
If BUFFER exists and shell process is running, just switch to BUFFER.

...
```

figure 16.1: Hilfe zur Funktion `shell`

Und siehe! Die Funktion macht prinzipiell schon alles, was du brauchst.

16.1.2 Der Makro

1. Beim interaktiven Aufruf mit Präfixargument `C-u M-x shell` wirst du nach dem Namen für den Buffer gefragt, das ist schon die halbe Miete.
2. Mit `C-c u z f` werden aktuelles Datum und Uhrzeit in günstigem Format für Dateinamen eingefügt:

   ```
   -20220614-175909
   ```

   Damit bekommst du die gewünschte Zeit. Das zugehörige Kommando erfährst du mit `C-h k C-c u z f`:

   ```
   C-c u z f runs the command ws-time-insert-filename-timestamp-string
   (found in global-map), which is an interactive autoloaded Lisp closure
   in 'ws-time.el'.

   It is bound to C-c u z f.

   (ws-time-insert-filename-timestamp-string &optional DATE-TO-USE)

   Insert DATE-TO-USE formatted as filename timestamp string at point.
   The time string format defined by
   'ws-time-filename-timestamp-format' is applied.
   When called interactively with prefix arg, DATE-TO-USE is
   prompted for.  Otherwise, the current date and time is used.
   ```

   Note: Als DATE-TO-USE kann man Dinge wie *yesterday* oder *tomorrow* oder auch *3 days ago* und *6 days* angeben. Das wäre doch schon eine nette Zusatzfunktion:

   - Mit negativem Argument wird ein Datum in der Vergangenheit benutzt.
   - Mit positivem Argument wird ein Datum in der Zukunft benutzt.

3. In zusammenfassender Kombination kannst du mit

   `C-x ( C-u M-x shell screenlog C-c u z f M-BSP BSP RET C-x ),`

   einen Makro definieren der deinen Anforderungen genügt.
4. Mit `M-x name-last-kbd-macro dax-screenlog-shell RET` und mif `M-x insert-keyboard-macro dax-screenlog-shell RET` gibts dann wie gehabt den Makro als Tastenvektorkommando. D.h., es ist interaktiv mit `M-x dax-screenlog-shell` ausführbar:

   ```
   (fset 'dax-screenlog-shell
      [?\C-u ?\M-x ?s ?h ?e ?l ?l return ?s ?c ?r ?e ?e ?n ?l ?o ?g ?\C-c ?u ?z ?f M-
   ↪backspace backspace return])
   ```

Damit sind die grundsätzlichen Anforderung erfüllt.

16.1.3 Die Funktion

Warning: Wenn du keine Lust hast, diese Vorarbeiten und Recherchen per eingebauter Hilfe zu betreiben, dann brauchst du nicht mit der Lisp-Programmierung anfangen. Das Ergebnis wäre ähnlich der Shell-Programmierung ohne Man-Pages. Das Rum-Googeln bringt im Ergebnis auch nur den Hinweis was man verwenden kann und dann für Genauers in der Hilfe suchen muss. Kann man sich ziemlich oft sparen und gleich in den lokal vorhandenen Beispielen nachsehen.

Note: Wenn du in den Verzeichnissen `/usr/share/emacs/`, `~/.emacs.def` `/usr/local/share/emacs/site-lisp/` mit `grep-find` nach `(shell[ )]` suchst, findest du z.B. alle Aufrufe der Funktion `shell`. Dabei wird dann schnell klar, wie man den Buffer-Namen beeinflusst.

Note: Nach dem Ausführen eines Kommandos mit `M-x`, kann man in der Liste der interaktiv ausgewerteten Befehle `C-x ESC ESC` nachschauen, wie die Aktion in einem Programm umgesetzt wird.

Was muss man haben

Wie die Makroerstellung gezeigt hat, werden für das fancy-schmanzy Kommando also die folgenden Funktionen benötigt:

- `shell`, Syntax (aus der Hilfe zur Funktion):

```
(shell &optional BUFFER)
```

- `ws-time-insert-filename-timestamp-string`, Syntax (aus der Hilfe zur Funktion):

```
(ws-time-insert-filename-timestamp-string &optional DATE-TO-USE)
```

Für einen eventuellen Präfix ist ein optionales Argument nötig.

Note: Ein Kommando kann im Gegensatz zu einer einfachen Funktion interaktiv ausgeführt werden. D.h., es ist mit `M-x` aufrufbar. Und mit `M-x global-set-key RET` kann es einer Tasstenfolge zugewiesen werden. Sowohl Kommandos als auch Funktionen werden mit `defun` definiert. Der Unterschied ist subtil. Ohne *Cut-Past-Modify* geht die Sache mit Sicherheit schief.

Welche Vorlage benutzt man?

Als Vorlage nimmt man am besten ein Kommando, das bereits ein optionales Argument unterstützt. `shell` ist in diesem Fall so ziemlich der offensichtlichste Kandidat `:-;`

Wo finde ich also die Definition von `shell`? Natürlich mit der Hilfe: `C-h f shell RET` (siehe figure 16.1). Dort ist der Ort der Definition als Hyperlink angegeben (`shell.el`). Also eifrig Link aktivieren, die Funktion kopieren und die Dokumentation schrumpfen. Dann noch alles, außer der Deklaration `interactive` fort schmeißen:

```
(defun shell (&optional buffer)
  "Run an inferior shell, with I/O through BUFFER (which defaults to `*shell*').
Interactively, a prefix arg means to prompt for BUFFER."
  (interactive
   (list
    (and current-prefix-arg
         (prog1
             (read-buffer "Shell buffer: "
                          ;; If the current buffer is an inactive
                          ;; shell buffer, use it as the default.
                          (if (and (eq major-mode 'shell-mode)
                                   (null (get-buffer-process (current-buffer))))
                              (buffer-name)
                            (generate-new-buffer-name "*shell*")))
           (if (file-remote-p default-directory)
               ;; It must be possible to declare a local default-directory.
               ;; FIXME: This can't be right: it changes the default-directory
               ;; of the current-buffer rather than of the *shell* buffer.
               (setq default-directory
                     (expand-file-name
```

(continues on next page)

(continued from previous page)

```
                          (read-directory-name
                           "Default directory: " default-directory default-directory
                           t nil))))))))
  ;; [ ... hier war mal was ... ]
  )
```

Umgang mit Klammerausdrücken

Ohne korrekte Anwendung der Hilfsmittel, ist Lisp eher im BDSM-Bereich angesiedelt.

Note: Lisp besteht vollständig aus geklammerten Ausdrücken (siehe S-expressions). Emacs bietet reichlich Unterstützung für deren Handhabung.

Die wichtigsten Tastenfolgen sind (`C-h b`, in den Hilfe-Buffer wechseln, dann `M-x occur RET sexp RET`):

```
C-M-f      forward-sexp
C-M-b      backward-sexp
C-M-SPC    mark-sexp
C-M-k      kill-sexp
C-x C-e    eval-last-sexp
```

und zusätzlich:

```
M-(        insert-parentheses
M-)        move-past-close-and-reindent
```

sowie (`C-h b`, in den Hilfe-Buffer wechseln, dann `M-x occur RET list$ RET`):

```
C-M-n      forward-list
C-M-p      backward-list
C-M-d      down-list
C-M-u      backward-up-list
```

Um einen Klammerausdruck zu löschen ist es also sinnvoller auf der öffnenden Klammer `C-M-SPC BSP` zu drücken (oder auch `C-M-k`, wenn es in den `kill-ring` soll), als mühsam per Cursor-Steuerung das Ende zu suchen.

Erst mal die absoluten Basics

Die `interactive` Deklaration ist viel zu komplex, daher wird sie erst mal auf `(interactive (list))` reduziert (`C-M-k` auf der öffnenden Klammer von `(and`):

```
(defun shell (&optional buffer)
  "Run an inferior shell, with I/O through BUFFER (which defaults to `*shell*').
Interactively, a prefix arg means to prompt for BUFFER."
  (interactive
   (list
   ))
  ;; [ ... hier war mal was ... ]
  )
```

Jetzt braucht das Kind noch den richtigen Namen, und es wird schon mal die Funktion `shell` aufgerufen. Der zukünftige Buffer-Name wird schon mal als lokale Variable `buffer-name` angelegt:

```
(defun dax-screenlog-shell (&optional buffer)
  "Run an inferior shell, with I/O through BUFFER (which defaults to `*shell*').
Interactively, a prefix arg means to prompt for BUFFER."
  (interactive (list))
  (let (buffer-name)
    (setq buffer-name nil)
    (shell buffer-name)
```

(continues on next page)

(continued from previous page)

```
  ))
;; (dax-screenlog-shell)
```

Jetzt macht das Kommando `dax-screenlog-shell` bereits das gleiche wie das Kommando `shell` ohne Präfixargument.

Butter bei die Fische

Als nächstes wird die Zusammenstellung des Buffer-Namens im Grundsatz umgesetzt. Der Name besteht aus einem Präfix, einem Bindestrich und dem Tag. Buffer ohne Dateizuweisung werden außerdem im Emacs mit einem führenden und abschließenden `*` gekennzeichnet. Der Aufruf von `shell` wird zum Testen erst mal auskommentiert. Der Rückgabewert ist das Ergebnis des letzten Ausdrucks, der ausgewertet wird. In diesem Fall ist das der Buffer-Name.

Die Funktion wird definiert, indem man **nach der letzten Kommentarzeile** `C-x C-e` drückt. Der Rückgabewert wird im Statusbereich angezeigt, wenn man am Ende der letzten Kommentarzeile `C-x C-e` drückt. Beim aktuellen Stand erscheint hier `"*screenlog-*"`.

```
(defun dax-screenlog-shell (&optional buffer)
  "Run an inferior shell, with I/O through BUFFER (which defaults to `*shell*').
Interactively, a prefix arg means to prompt for BUFFER."
  (interactive (list))
  (let (buffer-name prefix date)
    (setq prefix "screenlog")
    (setq date nil)
    (setq buffer-name (concat "*" prefix "-" date "*"))
    ;; (shell buffer-name)
    buffer-name
  ))
;; (dax-screenlog-shell)
```

Wir wissen bereits, dass `ws-time-insert-filename-timestamp-string` die aktuelle Zeit einfügt. Da das Einfügen aber nicht gewünscht ist, kann die Funktion nicht direkt verwendet werden. Wenn man sich zur Definition begibt, sieht man, dass das Ergebnis der Funktion `ws-time-filename-timestamp-string` mit `insert` eingefügt wird:

```
(insert (ws-time-filename-timestamp-string date-to-use))
```

Es liegt also nahe, `ws-time-filename-timestamp-string` direkt zu verwenden. `C-x C-e` am Ende jeder Kommentarzeile zeigt das Ergebnis der entsprechenden Auswertung:

```
;; "-20220614-195517"       <= (ws-time-filename-timestamp-string nil)
;; ("" "20220614" "195517") <= (split-string (ws-time-filename-timestamp-string nil) "-")
;; ("20220614" "195517")    <= (cdr (split-string (ws-time-filename-timestamp-string nil) "-
↪"))
;; "20220614"               <= (car (cdr (split-string (ws-time-filename-timestamp-string␣
↪nil) "-")))
```

Der letzte Ausdruck liefert wie gewünscht das Datum. und kann direkt in die Funktion `dax-screenlog-shell` eingefügt werden:

```
(defun dax-screenlog-shell (&optional buffer)
  "Run an inferior shell, with I/O through BUFFER (which defaults to `*shell*').
Interactively, a prefix arg means to prompt for BUFFER."
  (interactive (list))
  (let (buffer-name prefix date other-date)
    (setq prefix "screenlog")
    (setq date (car (cdr (split-string (ws-time-filename-timestamp-string other-date) "-"))))
    (setq buffer-name (concat "*" prefix "-" date "*"))
    ;; (shell buffer-name)
    buffer-name
  ))
;; (dax-screenlog-shell)
```

Die Prüfung des erzeugten Buffer-Namens zeigt, dass das erste Ziel damit erreicht ist. Daher kann der Aufruf von `shell` wieder aktiviert werden:

```
(defun dax-screenlog-shell (&optional buffer)
  "Run an inferior shell, with I/O through BUFFER (which defaults to `*shell*').
Interactively, a prefix arg means to prompt for BUFFER."
  (interactive (list))
  (let (buffer-name prefix date other-date)
    (setq prefix "screenlog")
    (setq date (car (cdr (split-string (ws-time-filename-timestamp-string other-date) "-"))))
    (setq buffer-name (concat "*" prefix "-" date "*"))
    (shell buffer-name)
    buffer-name
   ))
;; (dax-screenlog-shell)
```

Nach der Neudefinition präsentiert der Aufruf von `M-x dax-screenlog-shell RET` eine Shell mit dem gewünschten Namen. Weitere Aufrufe zeigen den existierenden Buffer wieder an und aktivieren ihn.

Das Sahnehäubchen

Dann wäre da noch das Präfixargument. In unserem Fall eine positive oder negative Zahl. Weiterhin muss Die Dokumentation angepasst werden und ein paar Optimierungen schaden auch nicht.

```
(defun dax-screenlog-shell (&optional offset-days)
  "Run an inferior shell with `*screenlog-<date>*' as buffer name.
Interactively, a numeric prefix arg means to add or subtract
OFFSET-DAYS to/from current date."
  (interactive (list (if current-prefix-arg
                         (prefix-numeric-value current-prefix-arg))))
  (let (buffer-name prefix date other-date)
    (cond
     ((and offset-days (< offset-days 0))
      (setq other-date (format "%d days ago" (abs offset-days))))
     ((and offset-days (> offset-days 0))
      (setq other-date (format "%d days" offset-days))))
    (setq prefix "screenlog")
    (setq date
          (cadr (split-string (ws-time-filename-timestamp-string other-date)
                              "-")))
    (setq buffer-name (concat "*" prefix "-" date "*"))
    (shell buffer-name)
    buffer-name))
;; (dax-screenlog-shell)
;; (dax-screenlog-shell -3)
;; (dax-screenlog-shell 5)
```

CHAPTER

SEVENTEEN

KLICKI-BUNTI-UND-DAS-FLIEWATÜÜT

Für alle, die meinen, dass sie es besser wüssten und wieder mal alle Tatsachen ignorieren wollen, gebe ich Folgendes zu bedenken.

– Und obwohl es lächerlich einfach ist und seit 1931 bekannt, hat es sich anscheinend immer noch nicht herumgesprochen, so dass man immer noch guten Gewissens haarsträubende Fehlentscheidungen treffen kann.

Es wäre wirklich schön, wenn in den Wollmilch-Sau-Diskussionen endlich mal die mathematischen Grundlagen berücksichtigen würden. Die sind völlig simpel und wer meint, sie widerlegen zu können, macht sich einfach nur lächerlich.

Grundsätzlich sind sowohl **Klicki-Bunti-Programme** als auch **Programmiersprachen** Mittel zur Problemlösung, die **per Algorithmus (Rechenvorschrift)** gesteuert werden.

Ein Algorithmus ist nichts magisches. Es ist einfach nur ein Rezept:

1. Programm Klicki-Bunti starten.
2. Datei Mümftel-Brümpf.xyz öffnen.
3. Menüpunkt `Vorwärts -> Seitwärts -> Nach Unten` auswählen.
4. Köpfe `Pitsche`, `Patsche`, `Peter` drücken.
5. Datei speichern unter Mümftel-Brümpf-kaputt.xyz speichern.

Das **Hauptklassifizierungsmerkmal** von Programmiersprachen ist die **Turing-Vollständigkeit**.

TL;DR: Die Erwartung, dass ein Klicki-Bunti-Programm **alle Probleme** lösen kann ist bestenfalls naiv.

Zitat Turing-Vollständigkeit – Wikipedia:

> Eine Maschine, die Turing-vollständig ist, kann jede Berechnung, die irgendein Computer ausführen kann, ebenso ausführen und wird daher auch als universell programmierbar bezeichnet. Hierdurch ergibt sich jedoch weder eine Aussage über den Aufwand, ein bestimmtes Programm auf einer solchen Maschine zu implementieren, noch über die Zeit, die zur Ausführung benötigt werden würde.

Vereinfacht ausgedrückt, heißt das, dass man mit einer Programmiersprache **entweder alles** programmieren kann oder **nur einen Teil** der Probleme.

Praktisch eingesetzte Porgrammiersprachen sind **Turing-vollständig** (Javascript, Python, Perl, C++, C, Java, C#). Die meisten **Klicki-Bunti-Programme** sind **nicht Turing-vollständig**. **Excel** ist z.B. erst seit Dezember 2020 **Turing-vollständig**, hat aber **unüberwindliche Wartungsprobleme**.

Hilfsmittel	Problemlösung vollständig möglich	Quell-Code	wartbar
Programmiersprache	ja	ja	ja
Excel	seit 2020	ja	nur mit Dokumentation
Klicki-Bunti	nein	nur per Dokumentation	extrem schlecht

CHAPTER

EIGHTEEN

SHELL OPTION EVALUATION LOOP

A generalized option evaluation loop is implemented in snippet *sh_b.opt-loop*. This chapter describes the algorithm employed.

18.1 Shell Commmand Grammar

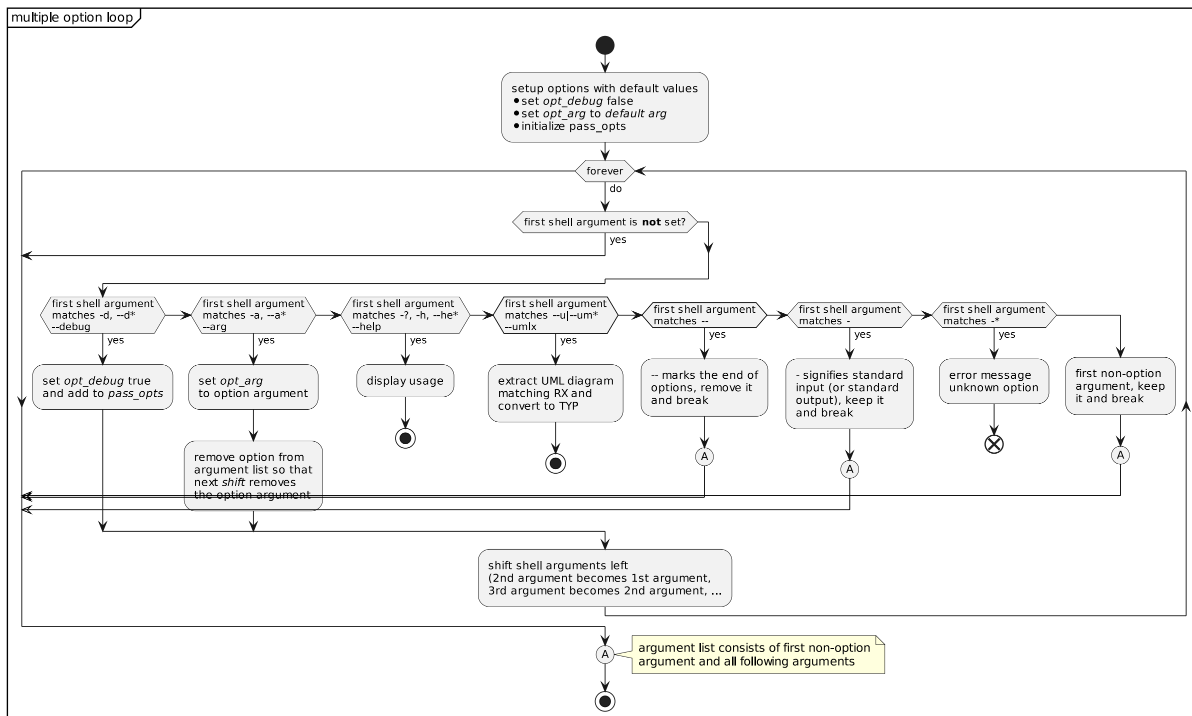
BNF for simple scripts:

```
command: command-name
       | command-name options
       | command-name arguments
       | command-name options arguments

options : option
        | option options

option: option-symbol
      | option-symbol argument
```

BNF for script implementing a command shell:

```
command-shell: command-shell-name
             | command-shell-name options
             | command-shell-name command
             | command-shell-name options command
```

18.2 Option Loop Activity Diagram



18.3 Option Loop Template

```
# ::here::
opt_debug=0
opt_arg='default arg'
pass_opts=
while :
do
    test x${1+set} = xset || break
    case "${1}" in
    ## (progn (forward-line 1) (snip-insert "sh_b.opt-option" t t "sh" " --key skip_for_
↪update --key opt-minimalx") (insert "\n"))
    # Defining opt_short as `o`, opt_long as `option`, opt_arg as `ARG` => creates option: -o,
↪ --option ARG; the value `::fillme::` is considered blank, it does not have to be removed
    # (let ((opt_short "::fillme::") (opt_long "::fillme::") (opt_arg "::fillme::")) (forward-
↪line 2) (snip-insert "sh_b.opt-option" t t "sh" (concat " --key opt-minimalx --key skip_for_
↪update --key opt_short --value " opt_short " --key opt_long --value " opt_long " --key opt_
↪arg --value " opt_arg)))
    # ::here::
    -d|--d|--de|--deb|--debu|--debug) # --debug
        opt_debug=1
        pass_opts="${pass_opts}"' --debug'
        ;;
    -a|--a|--ar|--arg) # --arg OPT-ARG
        opt_arg="${2-}"
        test x"${2+set}" = xset && shift
        pass_opts="${pass_opts}"' --arg '"${opt_arg}"
        #pass_opts="${pass_opts}"' --arg '"$( single_quote_enclose "${opt_arg}" )"
        ;;
    -\?|-h|--h|--he|--hel|--help) # --help
        usage; exit 0
        pass_opts="${pass_opts}"' --help'
        ;;
    --u|--um|--uml|--umlx) # --umlx RX[:TYP]
        opt_umlx="${2-}"
        test x"${2+set}" = xset && shift
```

(continues on next page)

(continued from previous page)

```
        opt_umlx_match="$( printf "%s\n" "${opt_umlx}" | ${SED__PROG-sed} 's,^\([^:]*\)\(:\([^
↪:]*\)\)*.*,\1,' )"
        opt_umlx_type="$( printf "%s\n" "${opt_umlx}" | ${SED__PROG-sed} 's,^\([^:]*\)\(:\([^
↪:]*\)\)*.*,\3,' )"
        out_filt_conv ()
        {
            _type="${1}"
            case "${_type}" in
            ## (progn (forward-line 1) (snip-insert "sh_b.case_patterns_for_actions" t t "sh"
↪" --key actions --value '::fillme::'") (insert ""))
            v|vi|vie|view) _type='svg';;
            esac
            if test -z "${_type}"
            then
                cat
            else
                plantuml -t"${_type}" -pipe
            fi
        }
        out_filt_disp ()
        {
            _type="${1}"
            case "${_type}" in
            ## (progn (forward-line 1) (snip-insert "sh_b.case_patterns_for_actions" t t "sh"
↪" --key actions --value '::fillme::'") (insert ""))
            v|vi|vie|view) _type='view';;
            esac
            if test "${_type}" != view
            then
                cat
            else
                display
            fi
        }
        (
        if test -n "${opt_umlx_match}"
        then
            line_diversion.py --match "${opt_umlx_match}" "${prog_path-${0}}"
        else
            line_diversion.py "${prog_path-${0}}"
        fi
        ) \
        | out_filt_conv "${opt_umlx_type}" \
        | out_filt_disp "${opt_umlx_type}"
        exit 0
        #pass_opts="${pass_opts}"' --umlx '"${opt_umlx}"
        #pass_opts="${pass_opts}"' --umlx '"$( single_quote_enclose "${opt_umlx}" )
        ;;
    --) # end of options
        shift; break;;
    -) # stdin
        break;;
    -*) # unknown option
        echo >&2 "${prog_name-$( basename "${0}" )}: error: unknown option \`${1}'"
        exit 1
        break
        ;;
    *)
        break;;
    esac
    shift
done
```

18.4 Snippet for defining an option

`sh_b.opt-loop` contains shell comments with emacs lisp code for inserting an option definiton:

```
;; opt_short: o | opt_long: option | opt_arg: ARG => description: -o, --option  ARG ||␣
↪::fillme:: is considered blank, it does not have to be removed
(let ((opt_short "::fillme::")
      (opt_long "::fillme::")
      (opt_arg "::fillme::"))
  (forward-line 1)
  (snip-insert "sh_b.opt-option" t t "sh"
               (concat " --key opt_short --value " opt_short
                       " --key opt_long --value " opt_long
                       " --key opt_arg --value " opt_arg)))
```

The *::fillme::* tag is interpreted as if the parameter was blank. I.e., the *::fillme::* tags do not have to be deleted, only replaced as necessary.

18.4.1 Example options

No parameters

The following Emacs Lisp expression, which does not define any of `opt_short`, `opt_long` and `opt_arg`,

```
(progn
  (forward-line 1)
  (snip-insert "sh_b.opt-option" t t "sh"))
```

results in the following snippet shell command and output:

```
>>>snc --mode 'sh' --key 'filename' --value 'sh_b.opt-option' --process --replace '/home/ws/
↪snippets/sh_b.opt-option'
#
opt_=0
    ) #
        opt_=1;;
```

All parameters = ::fillme::

The following Emacs Lisp expression, which defines `opt_short` as *::fillme::*, `opt_long` as *::fillme::* and `opt_arg` as *::fillme::*,

```
(let ((opt_short "::fillme::")
      (opt_long "::fillme::")
      (opt_arg "::fillme::"))
  ;; ...
  )
```

results in the following snippet shell command and output:

```
>>>snc --mode 'sh' --key 'filename' --value 'sh_b.opt-option' --key opt_short --value␣
↪::fillme:: --key opt_long --value ::fillme:: --key opt_arg --value ::fillme:: --process --
↪replace '/home/ws/snippets/sh_b.opt-option'
#
opt_=0
    ) #
        opt_=1;;
```

opt_short = q, opt_long = quiet-day

The following Emacs Lisp expression, which defines `opt_short` as *q*, `opt_long` as *quiet-day* and `opt_arg` as *::fillme::*,

```
(let ((opt_short "q")
      (opt_long "quiet-day")
      (opt_arg "::fillme::"))
  ;; ...
  )
```

results in the following snippet shell command and output:

```
>>>snc --mode 'sh' --key 'filename' --value 'sh_b.opt-option' --key opt_short --value q --key␣
↪opt_long --value quiet-day --key opt_arg --value ::fillme:: --process --replace '/home/ws/
↪snippets/sh_b.opt-option'
#   -q, --quiet-day
opt_quiet_day=0
    -q|--q*) # --quiet-day
        opt_quiet_day=1;;
```

opt_short = q, opt_arg = INT

The following Emacs Lisp expression, which defines `opt_short` as *q*, `opt_long` as *::fillme::* and `opt_arg` as *INT*,

```
(let ((opt_short "q")
      (opt_long "::fillme::")
      (opt_arg "INT"))
  ;; ...
  )
```

results in the following snippet shell command and output:

```
>>>snc --mode 'sh' --key 'filename' --value 'sh_b.opt-option' --key opt_short --value q --key␣
↪opt_long --value ::fillme:: --key opt_arg --value INT --process --replace '/home/ws/
↪snippets/sh_b.opt-option'
#   -q  INT
opt_q=''
    -q) #  INT
        opt_q="${2-}"
        test x"${2+set}" = xset && shift;;
```

opt_short = q, opt_long = quiet-day, opt_arg = INT

The following Emacs Lisp expression, which defines `opt_short` as *q*, `opt_long` as *quiet-day* and `opt_arg` as *INT*,

```
(let ((opt_short "q")
      (opt_long "quiet-day")
      (opt_arg "INT"))
  ;; ...
  )
```

results in the following snippet shell command and output:

```
>>>snc --mode 'sh' --key 'filename' --value 'sh_b.opt-option' --key opt_short --value q --key␣
↪opt_long --value quiet-day --key opt_arg --value INT --process --replace '/home/ws/snippets/
↪sh_b.opt-option'
#   -q, --quiet-day  INT
opt_quiet_day=''
    -q|--q*) # --quiet-day INT
        opt_quiet_day="${2-}"
        test x"${2+set}" = xset && shift;;
```

opt_long = quiet-day, opt_arg = INT

The following Emacs Lisp expression, which defines `opt_short` as *::fillme::*, `opt_long` as *quiet-day* and `opt_arg` as *INT*,

```
(let ((opt_short "::fillme::")
      (opt_long "quiet-day")
      (opt_arg "INT"))

  ;; ...
  )
```

results in the following snippet shell command and output:

```
>>>snc --mode 'sh' --key 'filename' --value 'sh_b.opt-option' --key opt_short --value␣
↪::fillme:: --key opt_long --value quiet-day --key opt_arg --value INT --process --replace '/
↪home/ws/snippets/sh_b.opt-option'
#    --quiet-day   INT
opt_quiet_day=''
    --q*) # --quiet-day INT
        opt_quiet_day="${2-}"
        test x"${2+set}" = xset && shift;;
```

18.4.2 Option generator snippet

```
# ||<-snap->|| if defined opt-minimal
# ||<-snap->|| alias opt-minimal rem
# ||<-snap->|| alias opt-not-minimal skip
# ||<-snap->|| subst opt-minimal-x
# ||<-snap->|| subst comm_activity @|empty@ #@empty@a99
# ||<-snap->|| subst case_short_end ;;
# ||<-snap->|| subst case_long_end
# ||<-snap->|| else
# ||<-snap->|| alias opt-minimal skip
# ||<-snap->|| alias opt-not-minimal rem
# ||<-snap->|| subst opt-minimal-x x
# ||<-snap->|| subst comm_activity @|empty@ #@empty@a0
# ||<-snap->|| subst case_short_end
# ||<-snap->|| subst case_long_end ;;
# ||<-snap->|| fi !defined opt-minimal
# ||<-snap->|| alias sh_b.opt-option rem
# ||<-snap->|| if !defined sh_b.opt-option
# ||<-snap->|| if defined skip_for_new
# ||<-snap->|| alias sh_b.opt-option skip
# ||<-snap->|| fi defined skip_for_new
# ||<-snap->|| fi !defined sh_b.opt-option

# ||<-snap->|| rem setup defaults
# ||<-snap->|| subst have_opt no
# ||<-snap->|| default opt_short
# ||<-snap->|| subst dash_opt_short

# ||<-snap->|| subst opt_short_long_comma
# ||<-snap->|| subst opt_short_long_bar

# ||<-snap->|| default opt_long
# ||<-snap->|| subst opt_name
# ||<-snap->|| subst dash_opt_long
# ||<-snap->|| subst opt_long_first
# ||<-snap->|| subst opt_long_cases
# ||<-snap->|| subst star_opt_long
# ||<-snap->|| subst comm_opt

# ||<-snap->|| default opt_arg
# ||<-snap->|| subst have_arg no
# ||<-snap->|| subst opt_arg_sep

# ||<-snap->|| rem ignore ::fillme:: tags
```

(continues on next page)

(continued from previous page)

```
# ||<-snap->|| if eq opt_init ::fillme::
# ||<-snap->|| undef opt_init
# ||<-snap->|| fi
# ||<-snap->|| if eq opt_short ::fillme::
# ||<-snap->|| subst opt_short
# ||<-snap->|| fi
# ||<-snap->|| if eq opt_long ::fillme::
# ||<-snap->|| subst opt_long
# ||<-snap->|| fi
# ||<-snap->|| if eq opt_arg ::fillme::
# ||<-snap->|| subst opt_arg
# ||<-snap->|| fi

# ||<-snap->|| rem setup opt_short
# ||<-snap->|| if !eq opt_short @|empty@
# ||<-snap->|| subst have_opt yes
# ||<-snap->|| subst dash_opt_short -
# ||<-snap->|| if !eq opt_long @|empty@
# ||<-snap->|| subst opt_short_long_bar |
# ||<-snap->|| subst opt_short_long_comma ,@|space@
# ||<-snap->|| else
# ||<-snap->|| subst comm_opt @|empty@ # @||dash_opt_short@@||opt_short@
# ||<-snap->|| fi
# ||<-snap->|| fi

# ||<-snap->|| rem setup opt_long
# ||<-snap->|| if !eq opt_long @|empty@
# ||<-snap->|| subst have_opt yes
# ||<-snap->|| subst dash_opt_long --
# ||<-snap->|| subst star_opt_long *
# ||<-snap->|| subst comm_opt @|empty@ # @|dash_opt_long@@|opt_long@
# ||<-snap->|| define opt_name process
# ||<-snap->|| exec dump !process replace
printf "%s\n" '@opt_long@' | ${SED__PROG-sed} 's,[^0-9A-Za-z],_,g'
# ||<-snap->|| exec
# ||<-snap->|| define opt_name
# ||<-snap->|| define opt_long_first process
# ||<-snap->|| exec dump !process replace
printf "%s\n" '@opt_long@' | ${CUT__PROG-cut} -c 1
# ||<-snap->|| exec
# ||<-snap->|| define opt_long_first
# ||<-snap->|| subst pass_opt @dash_opt_long@@opt_long@
# ||<-snap->|| else
# ||<-snap->|| subst opt_name @opt_short@
# ||<-snap->|| subst pass_opt @dash_opt_short@@opt_short@
# ||<-snap->|| fi

# ||<-snap->|| trim right
# ||<-snap->|| define opt_option_templates
    # Defining opt_short as `o`, opt_long as `option`, opt_arg as `ARG` => creates option: -o,
↪ --option ARG; the value `::fillme::` is considered blank, it does not have to be removed
    # (let ((opt_short "::fillme::") (opt_long "::fillme::") (opt_arg "::fillme::")) (forward-
↪line 2) (snip-insert "sh_b.opt-option" t t "sh" (concat " --key opt-minimal@opt-minimal-x@ -
↪-key skip_for_update --key opt_short --value " opt_short " --key opt_long --value " opt_
↪long " --key opt_arg --value " opt_arg)))
# ||<-snap->|| define opt_option_templates
# ||<-snap->|| if !defined skip_for_update
# ||<-snap->|| default opt_option_update
# ||<-snap->|| if !defined skip_for_new
# ||<-snap->|| if eq have_opt yes
# ||<-snap->|| subst opt_option_update ## **CONDSIDER UPDATING THE OPTION GENERATOR TEMPLATE**
# ||<-snap->|| fi eq have_opt yes
# ||<-snap->|| fi !defined skip_for_new
@opt_option_update@
# ||<-snap->|| trim right
    ## (progn (forward-line 1) (snip-insert "sh_b.opt-option" t t "sh" " --key skip_for_
↪update --key opt-minimal@opt-minimal-x@") (insert "\n"))
@opt_option_templates@
@opt_option_update@
# ||<-snap->|| trim right
# ||<-snap->|| fi !defined skip_for_update
```

(continues on next page)

(continued from previous page)

```
# ||<-snap->|| if defined skip_for_update
# ||<-snap->|| if !defined skip_for_new
# ||<-snap->|| if !eq have_opt yes
@opt_option_templates@
# ||<-snap->|| fi !eq have_opt yes
# ||<-snap->|| fi !defined skip_for_new
# ||<-snap->|| fi defined skip_for_update
# ||<-snap->|| sh_b.opt-option
# ||<-snap->|| undef skip_for_update
# ||<-snap->|| subst skip_for_new

# ||<-snap->|| rem setup opt_long_case_patterns
# ||<-snap->|| define opt_long_case_patterns process
# ||<-snap->|| exec !process replace
python -c '
from __future__ import print_function
import sys;
for arg in sys.argv[1:]:
    output = []
    _indx = 3
    while True:
        output.append(arg[:_indx])
        if not arg[_indx:]:
            print("|".join(output))
            break
        _indx += 1
' '@dash_opt_long@@opt_long@'
# ||<-snap->|| exec
# ||<-snap->|| define opt_long_case_patterns

# ||<-snap->|| rem setup opt_arg
# ||<-snap->|| if eq opt_short @|empty@
# ||<-snap->|| if eq opt_long @|empty@
# ||<-snap->|| subst opt_arg
# ||<-snap->|| fi
# ||<-snap->|| fi
# ||<-snap->|| if defined opt_arg
# ||<-snap->|| if !eq opt_arg @|empty@
# ||<-snap->|| subst have_arg yes
# ||<-snap->|| subst opt_arg_sep @|space@
# ||<-snap->|| fi
# ||<-snap->|| fi

# ||<-snap->|| rem setup initial option value
# ||<-snap->|| if !defined opt_init
# ||<-snap->|| if eq have_arg yes
# ||<-snap->|| subst opt_init ''
# ||<-snap->|| else
# ||<-snap->|| subst opt_init @|empty@0
# ||<-snap->|| fi
# ||<-snap->|| fi

# ||<-snap->|| trim right
# ||<-snap->|| if eq have_opt yes
# ||<-snap->|| rem insert description, initialization, parser
# ||<-snap->|| rem #   @dash_opt_short@@opt_short@@opt_short_long_comma@@dash_opt_long@@opt_
↪long@@opt_arg_sep@@opt_arg_sep@@opt_arg@
#   | @dash_opt_short@@opt_short@@opt_short_long_comma@@dash_opt_long@@opt_long@ |@opt_arg_
↪sep@@opt_arg@ | ::fillme:: |@comm_activity@
opt_@opt_name@=@opt_init@
    @dash_opt_short@@opt_short@@opt_short_long_bar@@opt_long_case_patterns@)@comm_opt@@opt_
↪arg_sep@@opt_arg@
# ||<-snap->|| if eq have_arg yes
        opt_@opt_name@="${2-}"
        test x"${2+set}" != xset || shift@case_short_end@
# ||<-snap->|| opt-not-minimal
        #pass_opts="${pass_opts}"' @pass_opt@ '"${opt_@opt_name@}"
        #pass_opts="${pass_opts}"' @pass_opt@ '"$( single_quote_enclose "${opt_@opt_name@}" )"
        @case_long_end@
# ||<-snap->|| opt-not-minimal
# ||<-snap->|| else
```

(continues on next page)

(continued from previous page)

```
        opt_@opt_name@=1@case_short_end@
# ||<-snap->|| opt-not-minimal
        pass_opts="${pass_opts}"' @pass_opt@'
        @case_long_end@
# ||<-snap->|| opt-not-minimal
# ||<-snap->|| fi
# ||<-snap->|| fi
# ||<-snap->|| sh_b.opt-option
```

ABBREVIATIONS

abbr see *Abbreviation*

GLOSSARY

Abbreviation As Wikipedia describes it[WPABBR]:

> An abbreviation (from Latin brevis, meaning short) is a shortened form of a word or phrase. It consists of a group of letters taken from the word or phrase. For example, the word abbreviation can itself be represented by the abbreviation *abbr.*, *abbrv.*, or *abbrev.*

INDEX

A

abbr, **81**
Abbreviation, **82**