~~((...))~~ ~~→ special~~

((... | ...)) → special / spaceless

((s1 s2 |; ...)) → s1, s2!

<< ... | ... >> → special / spaceful

<< s1 s2 |, ! >> s1, s2!

∅ [comment]

¥o = "os"

¥c = cou

¥s = [s/

S1 = Bottle

S2 = on the

S3 = wall

S4 = beer

¥b

¥c S1; ¥s S2; S3;

¥c S1; ¥s os S4; S2; S3;

¥c □ S1; ¥s □ of □ S4; □ S2; □ S3; □ ⇒ 26 chars

(LL) S1

strings to standard
(no uncompressing)

`...` → Nothing, already standard

q...q → Add spaces after ; IF unescaped & two chars before (that are valid sec chars)

Then replace q w/ ` (leading & trailing)

'...' → Every two chars, add a ; & a space
then replace leading ' & trailing '
w/ `

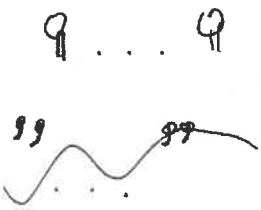
''...'' → Like above, but w/ no spaces

«...|...« → (1) split the format section into a list of expressions

(2) Every two chars, add an ,item from the format list & a space

(3) « → `

Normal strings - Takes standard chars & SCCs as is



Spaceful strings - Same as above but spaces after SCCs

Spaceless compressed strings - SCCs only, no spaces

spaceful compressed strings - like above, but with spaces

~~Speco~~

~~Smo.~~

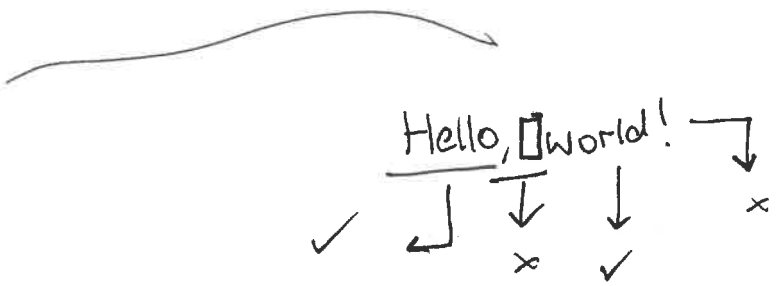
Special compressed strings

This needs a new page.

Differences between compressed & uncompressed strings

Points of contention:

* Spaces between SCCs
* String delimiters
* Length



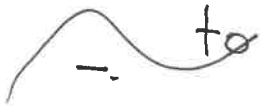
`s1; s2;!` → 9 + 2

New idea:
two types of strings

careless → ` ` → as is, no spaces between SCCs

careful → ` ` → spaces between SCCs

Conditional params



@ name _[else] | ... @

Tries to pop item from stack

& use that many params

else, uses given data!!

Adds n items into new array

e.g. [1, 2, 3]

@A; → [[1, 2, 3]]

@A_[*]|`n[]`(!1-(+)@

Returns 1 if all items in array are true

~~)AA *~~

@AA 1|!€c @^(!1-|@=[^1+])¥c=[1|0]@

keg+ BFL

@A;→@Z;

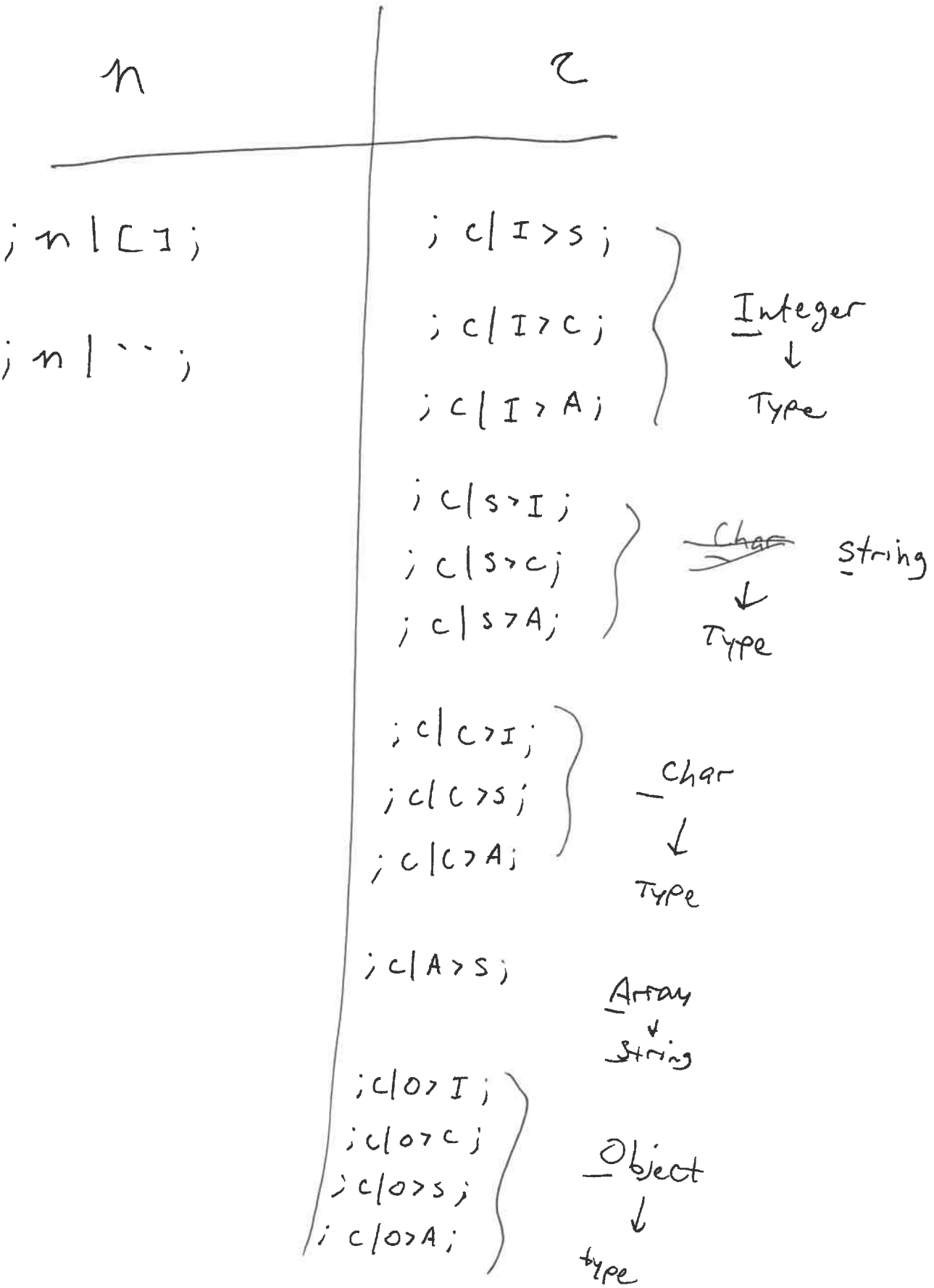
Section A - Array functions

@A*|`n[]i`(+)@ # Convert all items into one big # array

~~@AA*|~~

@AA 1|(+)@ # Array add (reduction)

@AB


n
c
;n|[I;
;n|``;
;c|I>s;
;c|I>c;
;c|I>A;
Integer
Type
;c|s>I;
;c|s>c;
;c|s>A;
Char
String
Type
;c|c>I;
;c|c>s;
;c|c>A;
Char
Type
;c|A>s;
Array
String
;c|o>I;
;c|o>c;
;c|o>s;
;c|o>A;
Object
type

The Uncompressor

* Used when either printing, comparing or

* Uncompressed upon start

* All strings turned into Standard strings (text only)

S1 = "Hello" S2 = "world"

S3 = "nice" ¥s = [s|""] → default s

Eg

Before

`Hello`

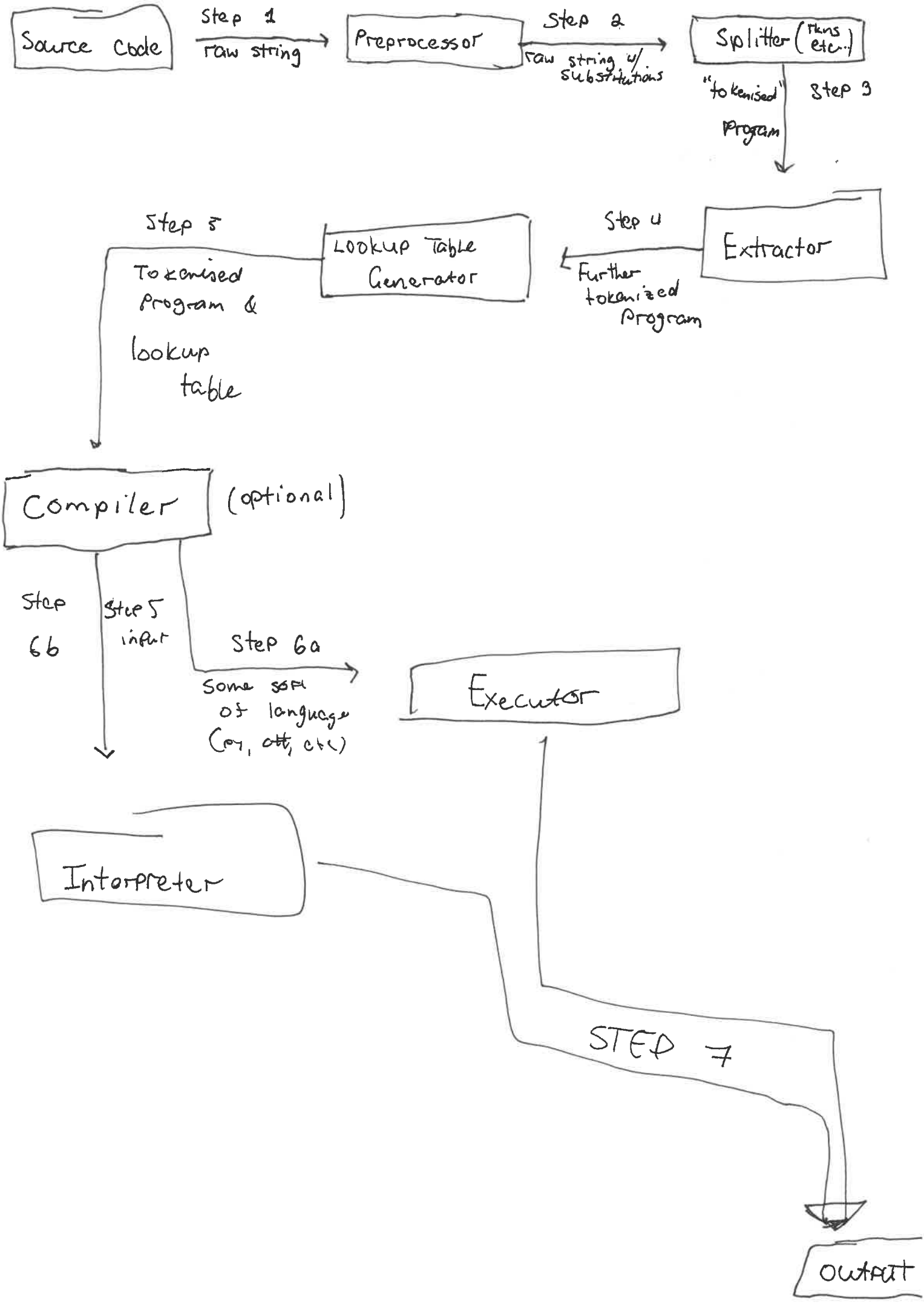
`S1;`

`S1;¥s`

After

`Hello`

`Hello`


Source Code
Step 1
raw string
Preprocessor
Step 2
raw string w/ substitutions
Splitter (runs etc..)
"tokenised" program
Step 3
Extractor
Step 4
Further tokenized Program
Lookup Table Generator
Step 5
Tokenised program & lookup table
Compiler (optional)
Step 6b
Step 5 input
Step 6a
Some sort of language (C++, ASM, etc)
Executor
Interpreter
STEP 7
output

A Library

Convert main stack items into an array

@ A _[A] | `n[]`(!1-/+) @

Add all items in array

@ AA 1 | "\.."(+) @

@ AA 1/ (!1-/ +)@

Convert all items in array into binary representation
if possible

@ AB 1 |

[3, 4] → ✓

[2, 0] → ✗

[3] → ✓

NO.

Structure extractor - kegt

```
def is_block(source):
    '''
    Given a source, this function will try to extract
    the data from an is statement "[expr_true/expr_false]"
    and place it in a dictionary"
    '''
```

Interpreter Parts

Phase 1 → (Substitution)

- The preprocessor → Not too much of a Syntax check (very light strings & "{"s)
  - ↙ Comes along and replaces references to the SSL (& ...) with the corresponding sequence
  - Doesn't replace:
    - occurances in strings
    - escaped &'s

- The uncompressor ↙ Replaces SCCs in strings
- The Tokenizer
  - ↙ Splits source into "tokens"
  - ↘ Literal string Command Sn...

(1) The Preprocessor:

```
Begin Preprocess(source)
    processed = ""
    string_mode = escaped = false
    For each character in source
        If string mode
        If character is "\" or "q" or "«"  //or any other string type
            If escaped
                escaped = False
                Processed += character

            Else
                String_mode = toggle(string_mode)
                processed += Character

        Else, If Character is "\"
```

A second look @ string compression...

Examples

3 letter SCCs	2 letter SCCs	1 letter SCCs
`eax;` → eax	`\eax;` → e ax	`le\ax;` → eax
`eax\;` → eax;	`ax;` → ax	`ea\x;` → eax;
`abcd;` → a bcd	`ax\;` → ax;	`x;` → x
		`e\ax;` → ea x
		`x\;` → x;

~~<< S1 S2 S3~~

Goal: $1 S2 ; S3 S4

<< S1 S2 $3 $4 | ; <<

<< S1 S2 S3 S4 ! ; <<

~~; *; << S1 S2 *; S3 S4 <<~~ << S1 S2 ¿ ; S3 S4 <<

NEW IDEA!!!

Pre-defined Constants!

character command: ¿

Usage:

¿name

a-z → as is A-Z → as is

0 → 100
1 → 10
2 → 256
3 → 32
4 → 64
5 → 25
6 → 1000
7 → 500
8 → 125
9 → 10,000
~~10~~

all other chars → as is

well, maybe not...

hmm...

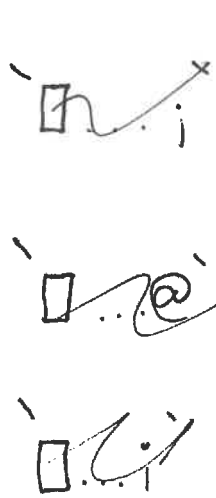
Object Strings

The ↑ undervalued backbone of keg+
in the past

A brief overview:

- contained within ~~standard~~ ~~strings~~ ;s
- * To be implemented in the interpreter
- * Allows for non-keg constructs to easily be made
- ** NOT preprocessed in anyway
- * Allows for BFL to be 100% keg+
- * Needs own library / file

Design



data

; □|...;

operative command

; n|[I ;

; c|S>I;

; n|` `;

; f|r.1;

; f|r.+;

To ambitious

```
1   Begin Standard (String)
2       create var result & Store an empty string
3       If String → type is "," then
4           create var code & store an empty string
5           Create var escaped & Store False
6           Append "," to result
7           For each char in string → data (reversed) do
8               If escaped then
9                   Toggle escaped
10                  Continue
11              Otherwise, If char is ";" then
12                  If code → length is 2 then
13                      Append scc → code (reversed) to result
14                  Set code to an empty string
15              otherwise, Is Char is "\" then
16                  Set escaped to True
17                  continue
18
19              If code → length is 2 then
20                  Append code[0] to result
                    Set code to code[1] + char
21              ~~Return result → reversed~~
22
23          Otherwise
24              Append Char to code
```

Form

;C|...1;

Where:

C is the command

...1 is the body

↓

h → new object ← stacks ;n|[];
← dicts ;n|{};
← strings ;n|"";
← files ;n|file;

c → cast (next page)

f → file ← open ;f|o; "img" → text
← read ;f|r.i; ;f|r.t;
← write/append ;f|w; ;f|a; ;f|ab; ;f|wb;
close

s → system
← etc...
← version ;s|version;
← operating sys ;s|platform;

def extract

"a $ b" → [{name: cmd, data: "a"},
{name: cmd, data: "$"},
{name: cmd, data: "b"}]

"[1 | 0]" → [{name: "if", data: {
True: [{name: cmd, data: "1"}]
False: [{name: cmd, data: "0"}]}]

"(0>>99 | ip | &ip %ip)" → ~~[{em~~

[{name: for, data: {
count: [{name: cmd, data: "0"},
{name: cmd, data: ">>"},
{name: cmd, data: ":"},
{name: cmd, data: "99"}

structure EXTRACTOR

Goal : A modular single function to return all data parts of a structure :.

[onTrue|onFalse] (count|var|code) {cond|type|code}

@name count[.] | code @ . But also, parse in a sense

Assumes prog is balanced

```
def extract(source):
    """
    final = []
    data = { }
    temp = ""
    structures = [] # structure stack
    current = ""
    for i in range(len(source)):
        char = source[i]
        if char not in "[](){}`" or escaped
```

OVERVIEW

Standard char

```
{"name": "char",
 "data": ...
}
```

If

```
{name: "if",
 data: {
        "true": ...,
        "false": ...
       }
}
```

For

Special strings

« S1S2S3 }4 |, !, ! LL

S1, S2! S3, S4!

$$\ll S1\,S2\,S3 \ldots SN \mid C_1 C_2 C_3 \ldots C_N \ll$$

$$S1C_1 \square S2C_2 \square S3C_3 \square \ldots S(N)C_N$$

$$\ll S1S2S3 \ldots SN \mid C_1 C_2 C_3 \ldots C_{N-1} \ll$$

$$S1C_1 S2C_2 S3C_3 \ldots S(N-1)_{C_{N-1}} SN$$

$$\ll S1S2S3 \ldots SN \mid C_1 C_2 C_3 \ll$$

$$S1C_1 \; S2C_2 \; S3C_3 \ldots SN$$

$$\ll S1S2S3 \ldots SN ! C_1 C_2 C_3 \ldots C_N \ll$$

$$S(C_N \; S2C_{N-1} \; S3C_{N-2} \ldots S(N-2)C_3 \; S(N-1)C_2 \; SNC_1$$

`¥x S1;¥s S2; S3; S4;`

1 2 3 4 5 6

`¥x S1;¥s S2; S3; S4;` → 22
(26 autocomp)

`<< ¥x S1 S2 S3 S4 | ¥s os <<` → 20
(19 auto comp)

`<< ¥x S1 ¥s S2 S3 S4 | os <<` → 19
(18 autocomp)

case	space
var → scc	✓
scc → var	✗
var → var	✓
scc → scc	✓

~~`<< ¥x S1 ¥s S2 S3 S4 <<`~~

`<< ¥x S1 ¥s ¥o S2 S3 S4 <<` → 15
(14 au

SCC only

□ S1S2S3 □

□ is the delimiter.

Simply start @ delimiter & move through string until Second delimiter. If the amount of Chars is odd - error.

If the chars contain "|;'q«„*»" → Error.

Otherwise, Split into pairs & do a Simple lookup

n - strings

objects

Stacks → ;n|[]; → pushes an empty stack
Strings → ;n|""; → "" string
Files → ;n|file; → creates a new file
Ports → ;n|p(ip); → creates a new internet conn
~~dictionaries~~

Math obj → ;n|m[Property]; → creates a new math object

Stack objs

;n|[]; → empty

;n|[0,1,2]; → [1,2,3]

;n|["hello"]; → ["Hello"]

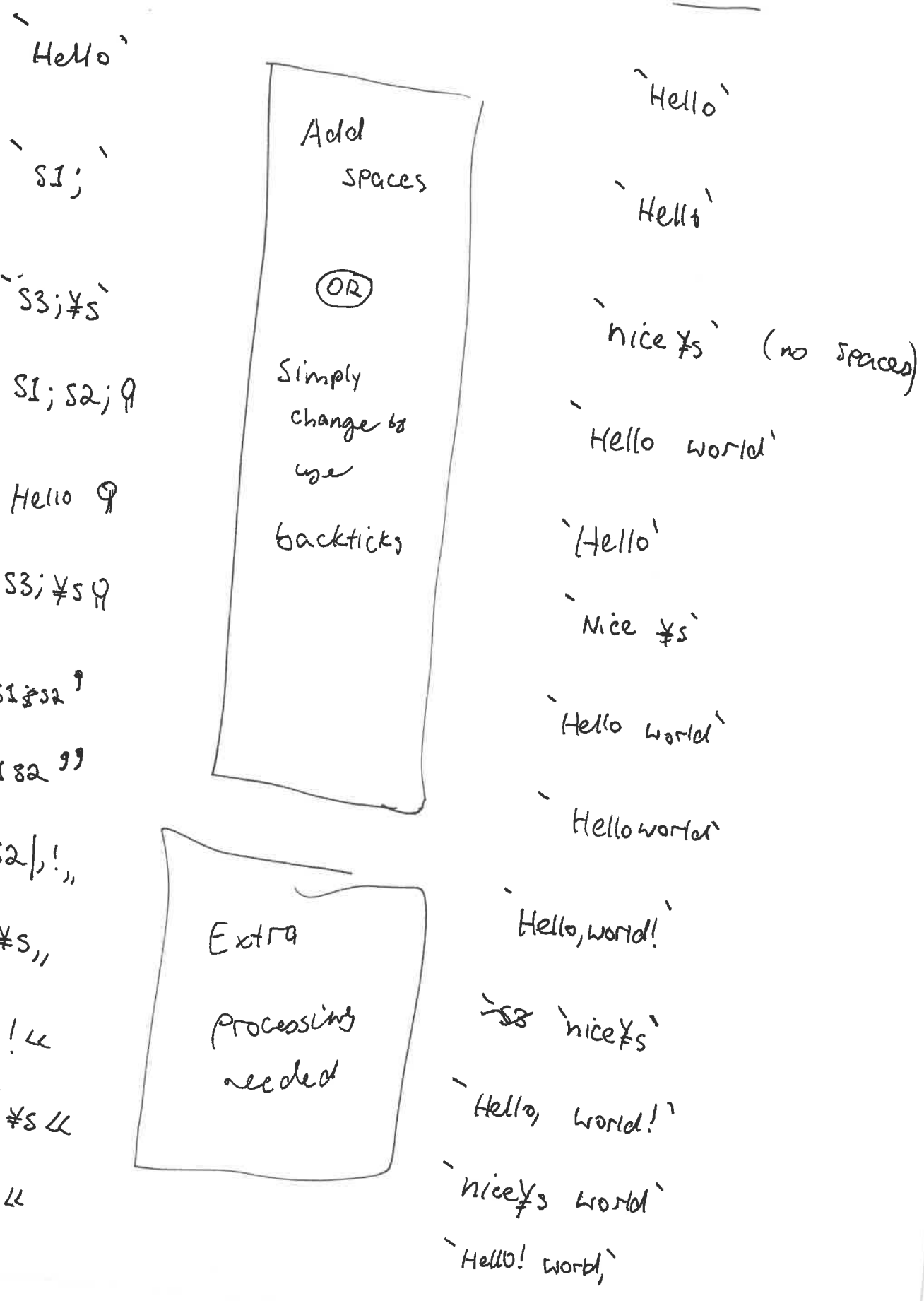
No.

The Uncompressor

* After preprocessor:

$1 = Hello
$2 = world $3 = nice

Before → After



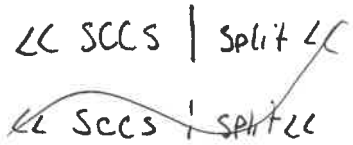
abcdefgh 8

[0, 2, 4, 6]

[s[i:i+2] for i in range(0, len(s), 2)]

s[0:2]

s[2:4]



codes [] joiners

The Preprocessor → Dael

* Implements the SSL (<char>)

Draft 1

```
def preprocess(source):
    final = ""
    in_string = False
    escaped = False

    for char in source:
        if char in "'" '"' "<":
            if escaped:
                escaped = False
            else:
                in_string = toggle(in_string)   # or Not in_string

    elif

END
```

"Failure because string type not stored"

```
else:
    if char == "&":
        if escaped:
            escaped = False
        elif string_type != "":
            SSL_needed = True
            Continue
    else:
        SSL_needed = False

final += char
```

⑦

```
if SSL_needed:
    final += SSL[char]
    SSL_needed = False
```