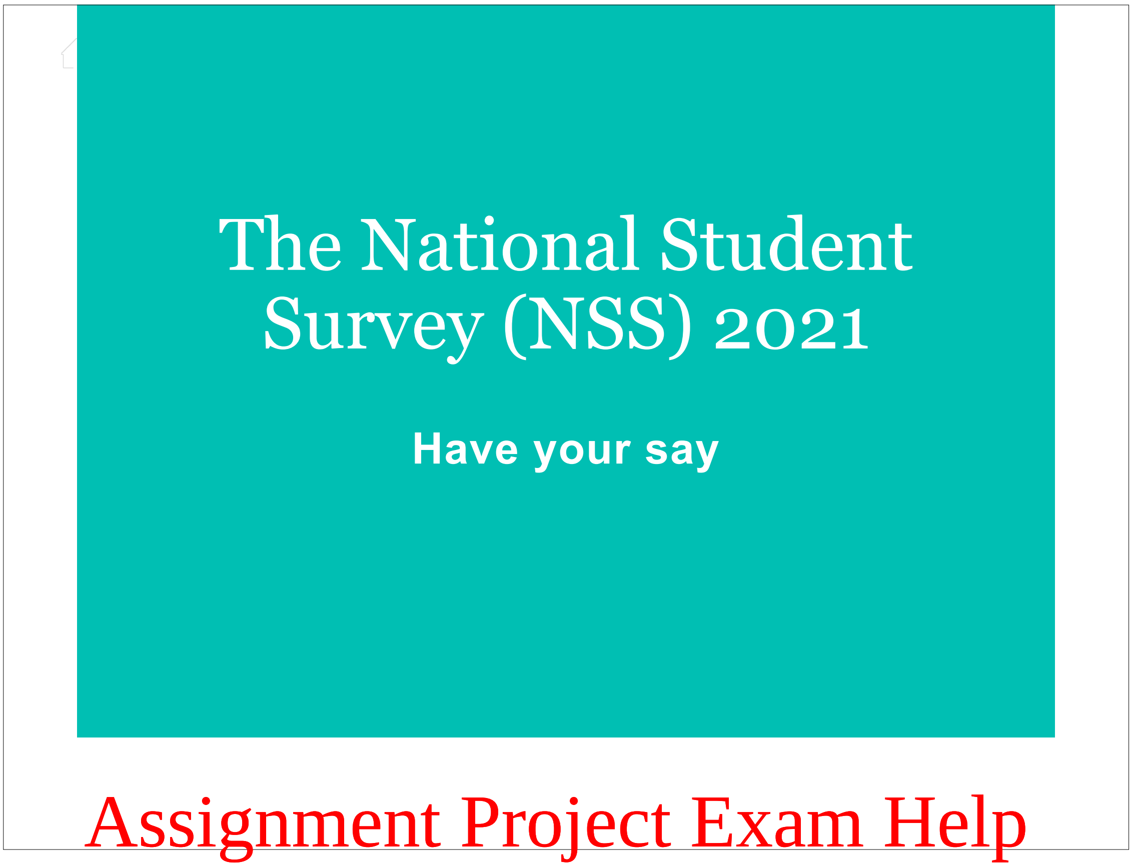
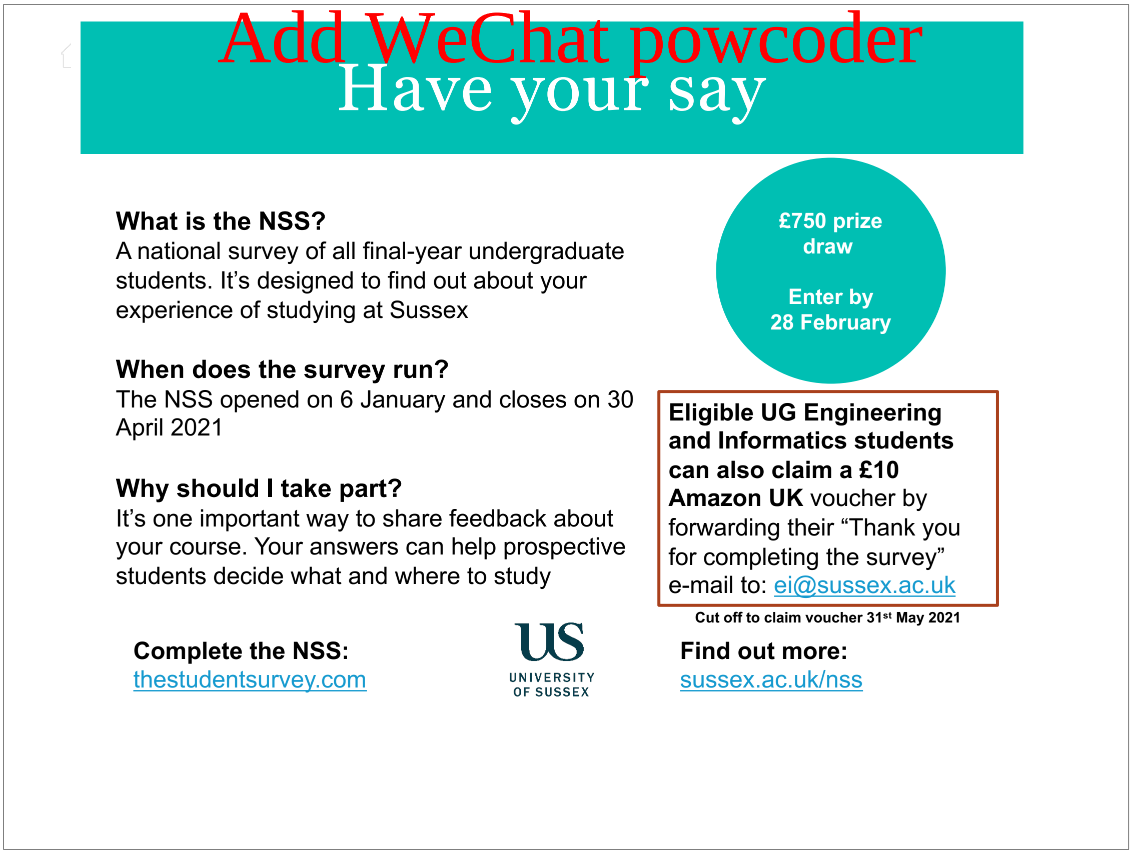


The National Student Survey (NSS) 2021

Have your say



Have your say

What is the NSS?
A national survey of all final-year undergraduate students. It's designed to find out about your experience of studying at Sussex

When does the survey run?
The NSS opened on 6 January and closes on 30 April 2021

Why should I take part?
It's one important way to share feedback about your course. Your answers can help prospective students decide what and where to study

£750 prize draw

Enter by 28 February

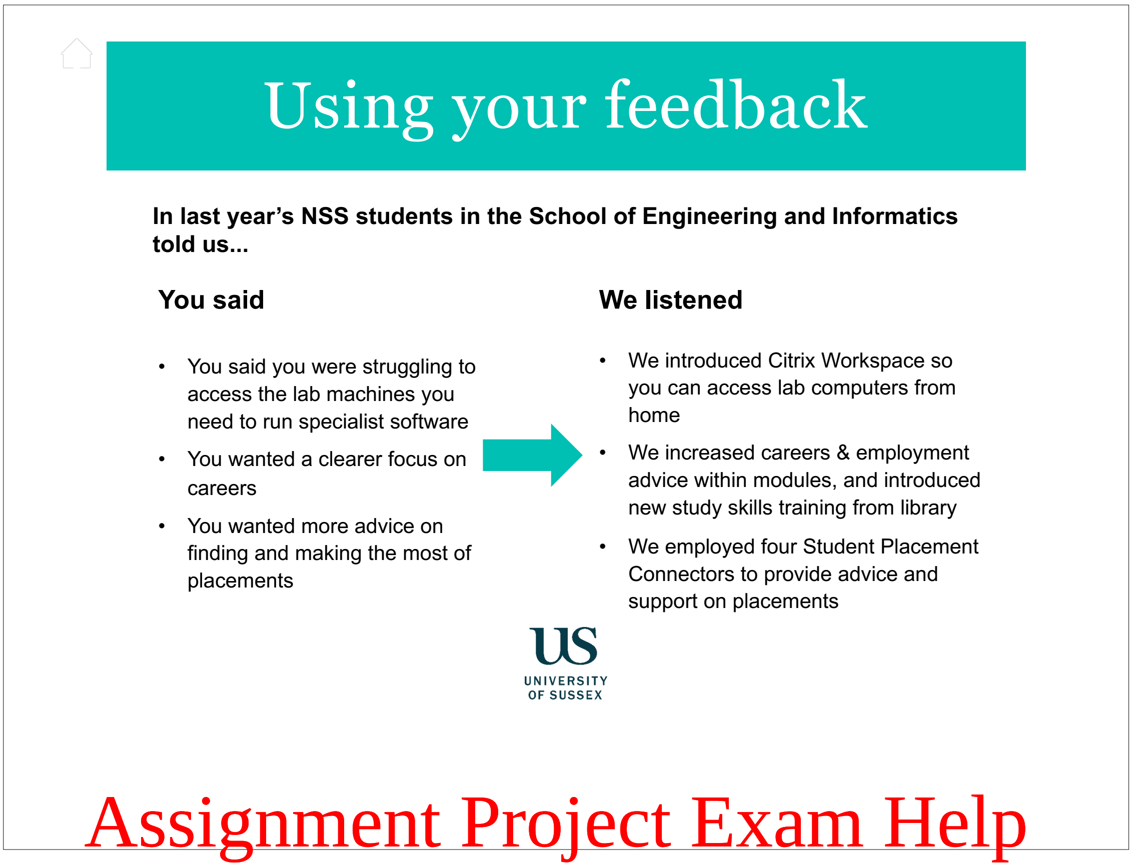
Eligible UG Engineering and Informatics students can also claim a £10 Amazon UK voucher by forwarding their "Thank you for completing the survey" e-mail to: ei@sussex.ac.uk

Cut off to claim voucher 31st May 2021

Complete the NSS:
thestudentsurvey.com

US
UNIVERSITY OF SUSSEX

Find out more:
sussex.ac.uk/nss



Using your feedback

In last year's NSS students in the School of Engineering and Informatics told us...

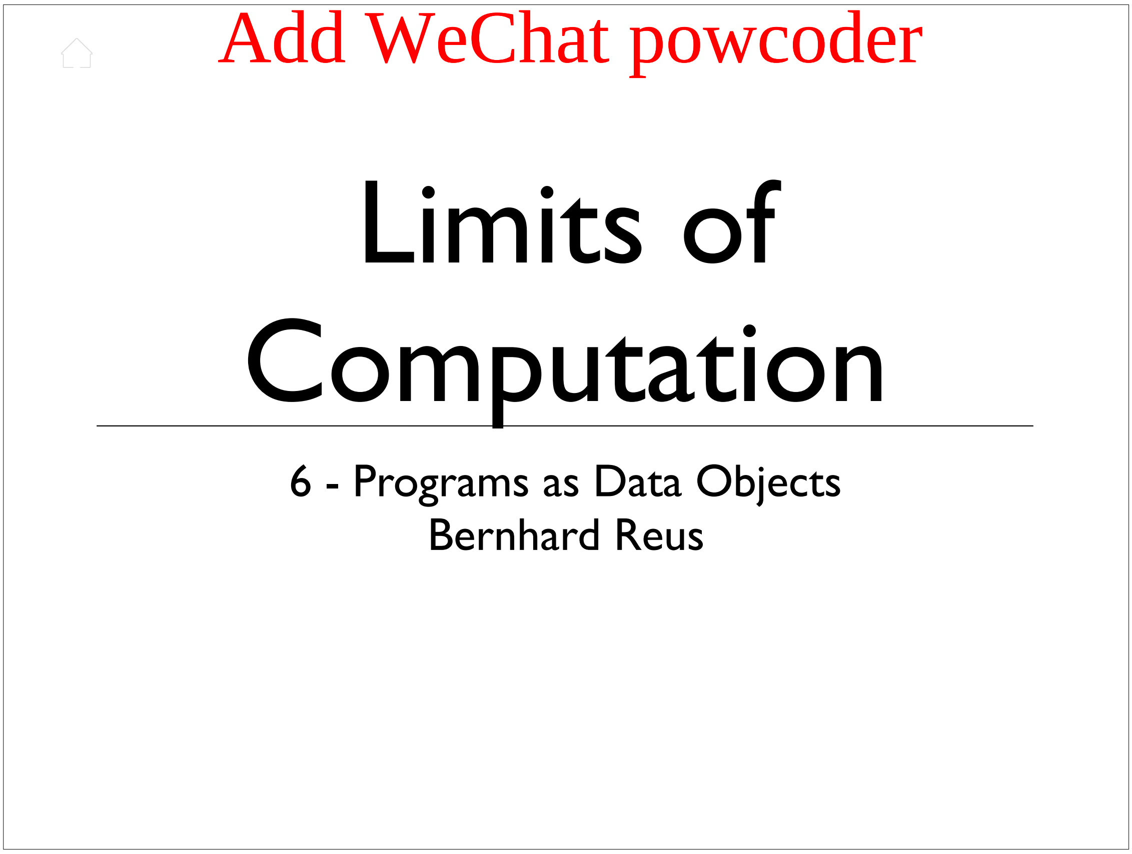
You said

- You said you were struggling to access the lab machines you need to run specialist software
- You wanted a clearer focus on careers
- You wanted more advice on finding and making the most of placements

We listened

- We introduced Citrix Workspace so you can access lab computers from home
- We increased careers & employment advice within modules, and introduced new study skills training from library
- We employed four Student Placement Connectors to provide advice and support on placements

US
UNIVERSITY OF SUSSEX



Limits of Computation

6 - Programs as Data Objects
Bernhard Reus

So far...

- "effective procedure" = WHILE-program
- introduced WHILE-language with binary tree data type ...
- ... that can also be viewed as a type of (arbitrary deeply) nested lists
- and extended WHILE for convenience

WHILE-programs as lists

THIS TIME

- We show how WHILE-programs can be ***data objects*** usable in another WHILE-program

```
[0,
 [[:=,1,[quote,nil]],
  [while,[var,0],
      [  [:=,1,[cons,[hd,[var,0]],[var,1]]],
         [:=,0,[tl,[var,0]]]
      ]
 ]],
 1]
```

A WHILE-program abstract syntax tree encoded as list

Programs as Input or Output

- Compiler
  program transformer which takes a program and translates it into an *equivalent* program, most likely in another language;
- Interpreter
  takes a program *and* its input data, and returns the result of applying the program to that input.
- Program Specialiser
  takes a *program with two inputs and* one data for one of the inputs and *partially evaluates* the program with the one given data producing a new program with one input only (more on that later).

Programming Languages

our notion, formally

Definition 6.1. A *programming language* L consists of

1. two sets: L-programs (the set of L-programs) and L-data (the set of data values described by the datatype used by this language)[1].
2. A function $[\![\_]\!]^{L}$: L-programs $\to$ (L-data $\to$ L-data$_\perp$) which maps L-programs into their semantic behaviour, namely a partial function mapping inputs to outputs, which are both in L-data.

PL with Pairing

Definition 6.2. A programming language $\texttt{L}$ defined as above *has pairing* if its data type, $\texttt{L}$-data, permits the encoding of pairs. For a general (unknown) language that has pairing we denote pairs (a,b), i.e. using parenthesis and a comma.

Does `WHILE` have pairing?

PL with Programs As Data

Definition 6.3. A programming language $\texttt{L}$ defined as above *has programs as data* if its data type, $\texttt{L}$-data, permits the encoding of $\texttt{L}$-programs. For a general (unknown) language that has programs as data the encoding of a program p is denoted $\ulcorner p \urcorner$.

The purpose of this session is
to show that `WHILE` has programs as data.

Programs as Data

- If language L has "***programs as data***" we can write compilers, interpreters, and specialisers in L.
- We want WHILE to have "*programs as data*".
- Thus we need a representation of WHILE programs as binary tree
- It is natural to use ***abstract syntax trees***

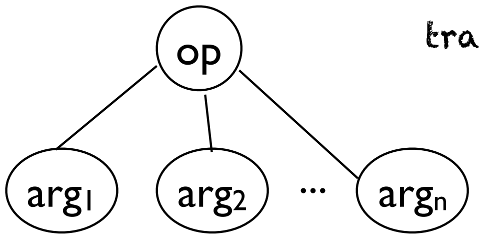
Interpreter

our notion, formally

Definition 6.4. Assume S has programs as data, S-data ⊆ L-data and L has pairing. An interpreter `int` for a language S written in L must fulfil the following equation for any given S-program p and $d \in$ S-data:

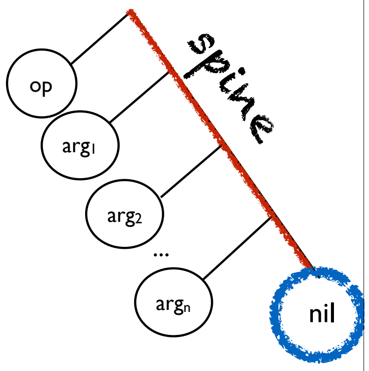
$$\boxed{[\![\mathtt{int}]\!]^{\mathtt{L}}(\ulcorner p \urcorner, d) = [\![p]\!]^{\mathtt{S}}(d)} \tag{6.1}$$

Abstract Syntax Trees as lists



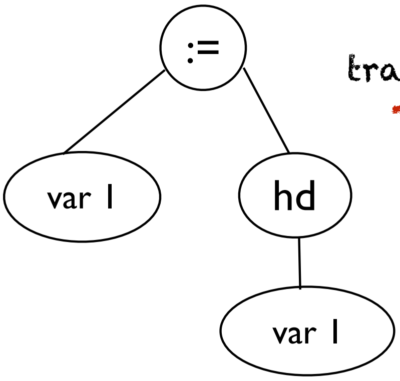
translates to →

[op, arg_1, arg_2,..., arg_n]



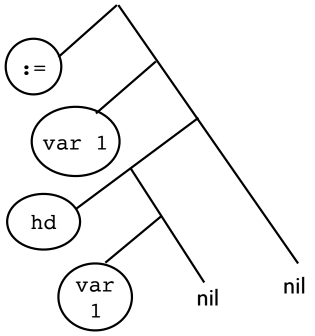
AST as list

`Y:= hd Y` (`Y` is 1st variable)



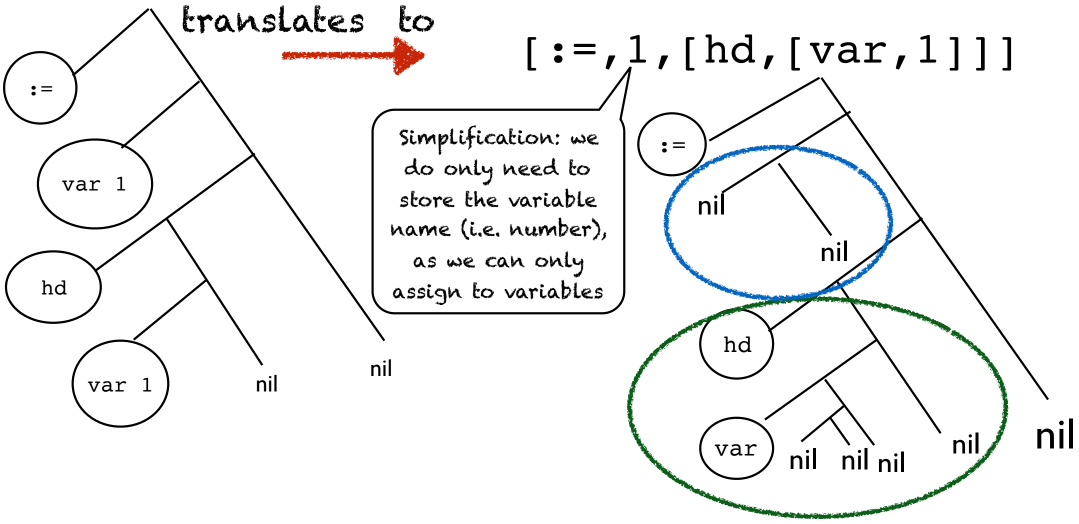
translates to →

`[:=,var1,[hd,var1]]`



var 1 needs to be unfolded as list too, see next slide

AST as list `Y:= hd Y` (`Y` is `var 1`)



`[:=,1,[hd,[var,1]]]`

Simplification: we do only need to store the variable name (i.e. number), as we can only assign to variables

What to do with var etc?

`[:=,1,[hd,[var,1]]]`

These are not yet trees/lists:

:= var hd

Answer: either introduce them as *additional atoms* or *encode them* (uniquely) as numbers.

Programs as data in `WHILE`

- We are now in a position to define more exactly how the list encoding of abstract syntax trees work.
- Lists are themselves encoded as binary trees.
- Let's go:

`WHILE` programs in $\mathbb{D}$

$$\ulcorner \texttt{progname read X } \{\texttt{S}\} \texttt{ write Y} \urcorner = [\,\textit{varnum}_{\texttt{X}}, \ulcorner\{\texttt{S}\}\urcorner, \textit{varnum}_{\texttt{Y}}\,]$$

commands

$$\begin{aligned}
\ulcorner \texttt{while E B} \urcorner &= [\texttt{while}, \ulcorner\texttt{E}\urcorner, \ulcorner\texttt{B}\urcorner] \\
\ulcorner \texttt{X} := \texttt{E} \urcorner &= [:=, \textit{varnum}_{\texttt{X}}, \ulcorner\texttt{E}\urcorner] \\
\ulcorner \texttt{if E B}_{\texttt{T}} \texttt{ else B}_{\texttt{E}} \urcorner &= [\texttt{if}, \ulcorner\texttt{E}\urcorner, \ulcorner\texttt{B}_{\texttt{T}}\urcorner, \ulcorner\texttt{B}_{\texttt{E}}\urcorner] \\
\ulcorner \texttt{if E B} \urcorner &= [\texttt{if}, \ulcorner\texttt{E}\urcorner, \ulcorner\texttt{B}\urcorner, [\,]]
\end{aligned}$$

$$\ulcorner \{\texttt{C}_1; \texttt{C}_2; \ldots; \texttt{C}_n\} \urcorner = [\ulcorner\texttt{C}_1\urcorner, \ulcorner\texttt{C}_2\urcorner, \ldots, \ulcorner\texttt{C}_n\urcorner]$$

expressions

$$\begin{aligned}
\ulcorner \texttt{nil} \urcorner &= [\texttt{quote}, \texttt{nil}] \\
\ulcorner \texttt{X} \urcorner &= [\texttt{var}, \textit{varnum}_{\texttt{X}}] \\
\ulcorner \texttt{cons E F} \urcorner &= [\texttt{cons}, \ulcorner\texttt{E}\urcorner, \ulcorner\texttt{F}\urcorner] \\
\ulcorner \texttt{hd E} \urcorner &= [\texttt{hd}, \ulcorner\texttt{E}\urcorner] \\
\ulcorner \texttt{tl E} \urcorner &= [\texttt{tl}, \ulcorner\texttt{E}\urcorner]
\end{aligned}$$

Example

```
reverse read X {
 Y:= nil;
 while X {
   Y:= cons hd X Y;
   X:= tl X
   }
}
write Y
```

```
X is var 0
Y is var 1
```

translate program into data

```
[0,
 [[:=,1,[quote,nil]],
  [while,[var,0],
      [ [:=,1,[cons,[hd,[var,0]],[var,1]]],
        [:=,0,[tl,[var,0]]]
      ]
 ]],
 1]
```

Programs-as-data in *hWhile*

- We can now write compilers, interpreters, specializers in `WHILE` using abstract syntax trees in list notation ("programs-as-data") instead of string representation.
- Thus we do not have to care about parsing programs.
- In *hwhile* (see Canvas) we can use the -u flag to produce this list representation:

```
hWhile -u reverse.while
```

```
[ 0
,
    [ [@:=, 1, [@quote, nil]]
    ,
        [ @while, [@var, 0]
        ,
            [ [@:=, 1, [@cons, [@hd, [@var, 0]], [@var, 1]]]
            , [@:=, 0, [@tl, [@var, 0]]]
            ]
        ]
    ]
, 1
]
```

hWhile uses @ to indicate special atoms

A note on hWhile output

- *hWhile* output by default is given as binary tree:

```
./hwhile add [3,4]
<nil.<nil.<nil.<nil.<nil.<nil.<nil.nil>>>>>>>
```

- *use flags to determine the "type" in which it is presented*

```
./hwhile -i add [3,4]
7
```

integer

```
./hwhile -l add [3,4]
[nil,nil,nil,nil,nil,nil,nil]
```

list of trees

```
./hwhile -li add [3,4]
[0, 0, 0, 0, 0, 0, 0]
```

list of integers

A note on `hWhile` output

- *There are more output formats, to see them all run:*

```
./hwhile -h
```

- *Look at this one, can you explain it?*

```
/hwhile -La  add [3,4]
@doWhile
```

`-La` ?

END

Next time:
A special interpreter