Limits of Computation

7 - A universal program (Self-interpreter)

Bernhard Reus

So far...

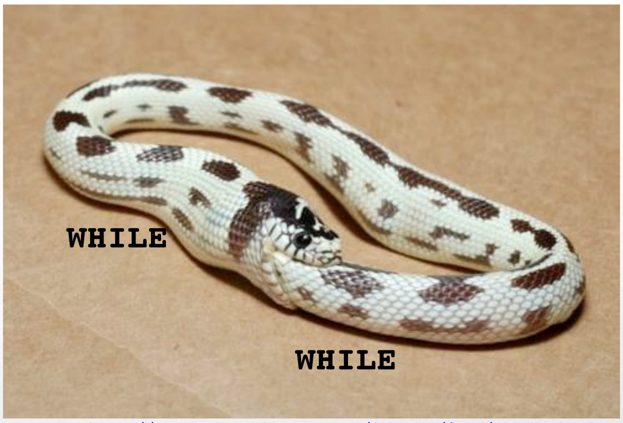
- ... we have learned the `WHILE`-language…
- ...that we have chosen to represent our notion of computation (to write “effective procedures”).
- We learned how to represent programs-as-data...
- ...so now we **can write interpreters**.

Eating your own tail?

- We look at a special form of interpreter: THIS TIME
- **self-interpreter**

  `WHILE`-interpreter written in `WHILE`

  → and first an `WH`^{1}`LE`-interpreter written in `WHILE`
- a very important concept for computability theory *(used later)*



http://www.strangedangers.com/content/item/158424.html

Compare to TMs

- Turing defined a "universal Turing machine" *U*
- that can take TM program description *D* and a word *W* as input on its tape
- and simulate the run of TM *D* with given input *W*
- *so U is a TM program which is an interpreter for TM programs*

a self-interpreter in TM

let's use `WHILE` instead!

Use of self-interpreter?

- in practice:
  "cheap" way to extend your programming language with extra features (interpret them in smaller language)
- in computability theory:
  we will explain this soon. Stay tuned!

First consider WH^{1}LE

- ...is like `WHILE`...
- ...but programs can only have ***one*** variable.
- simpler "memory management"
- Can we solve *more* problems with programs in `WHILE` than in WH^{1}LE?

Interpret $\texttt{WH}^1\texttt{LE}$ in `WHILE`

- Since it is simpler, we first look at an interpreter of $\texttt{WH}^1\texttt{LE}$ written in `WHILE`.
- Then we generalise to arbitrarily many variables and obtain a `WHILE`-interpreter in `WHILE`.

Recipe

Tree Traversal of ASTs

(with intermediate results)

NO RECURSION !

initialise tree and value stack to be empty
push tree (to be traversed) on tree stack
while tree stack not empty
  pop a tree t from tree stack
  if t is just an opcode o with arity n // *a marker*
  then pop n results $r_1,\ldots,r_n$ *from value stack*
    $r := o(r_1,\ldots,r_n)$ // compute intermediate result
    push r on value stack
  else // t proper tree
  if t's opcode has n arguments
  then push t's opcode on tree stack // (as marker!)
    push n subtrees of t on tree stack
  else // *o is leaf*
    compute result and push on value stack

order of arguments important

WH^{1}LE-interpreter

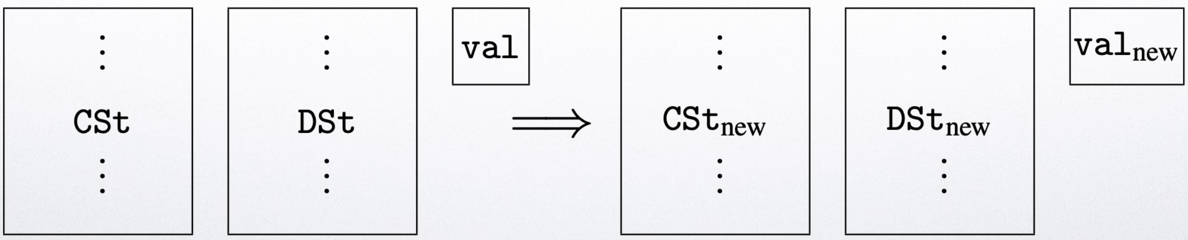
```
read PD  {                   (* input is a list [P,D] *)
  P := hd PD ;               (* P = [X,B,X] *)
  D := hd tl PD;             (* D input data *)
  B := hd tl P;              (* B is program block *)
  CSt := B;                  (* CSt is code stack  *)
                             (* initially  commands of B *)
  DSt := nil;                (* DSt is computation stack for *)
                            (* intermediate results *)
  val := D;                  (* D is initial value of variable *)
  state := [ CSt, DSt, val ];
                              (* wrap up state for STEP macro *)
  while CSt {                (* main loop for interpretation *)
    state := <STEP> state;(* loop body macro *)
    CSt := hd state          (* get command stack *)
    }
  val := hd tl tl state      (* get final value of variable *)
}
write val                    (* return value of the one variable *)
```

CSt is the code stack (code in list format),
DSt is the stack of intermediate values,
val contains value D of the one variable

Step Macro

performs tree traversal based on `CSt`, `DSt`, and `val`.

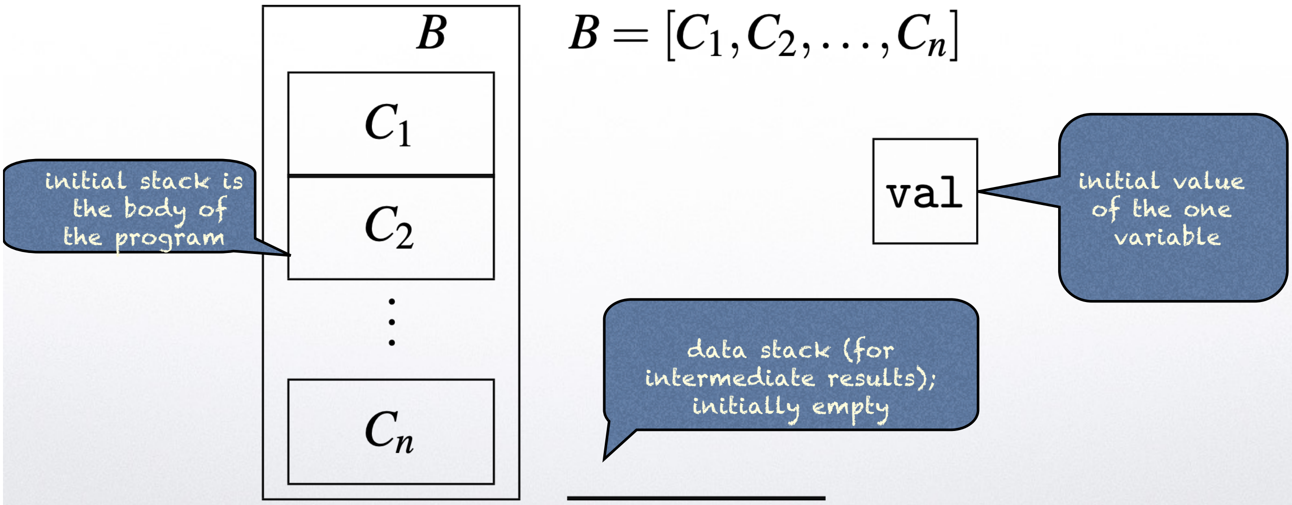
$$[\texttt{CSt},\texttt{DSt},\texttt{val}] \Rightarrow [\texttt{CSt}_{\text{new}},\texttt{DSt}_{\text{new}},\texttt{val}_{\text{new}}]$$



Initial set-up

```
state := [ CSt, DSt, val ];
```

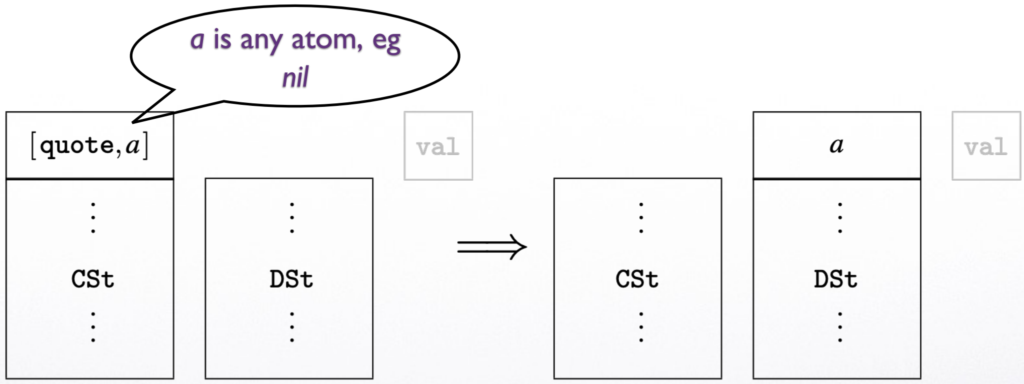
$$B = [C_1, C_2, \ldots, C_n]$$



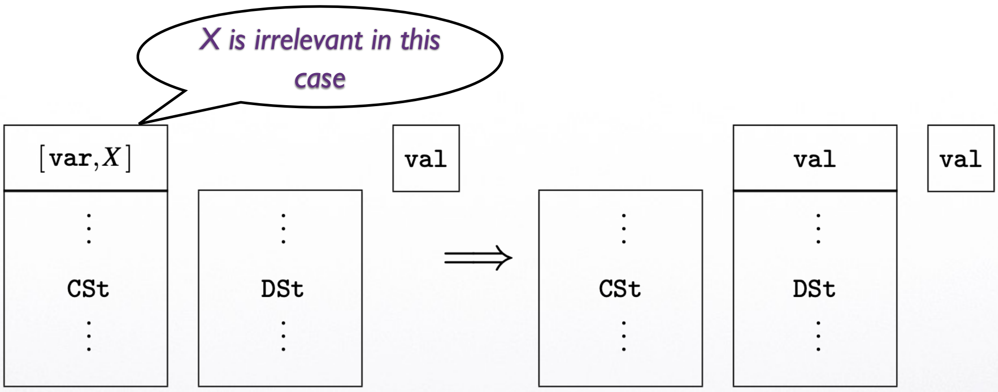
AST Leaves

(expressions without arguments)

Atoms



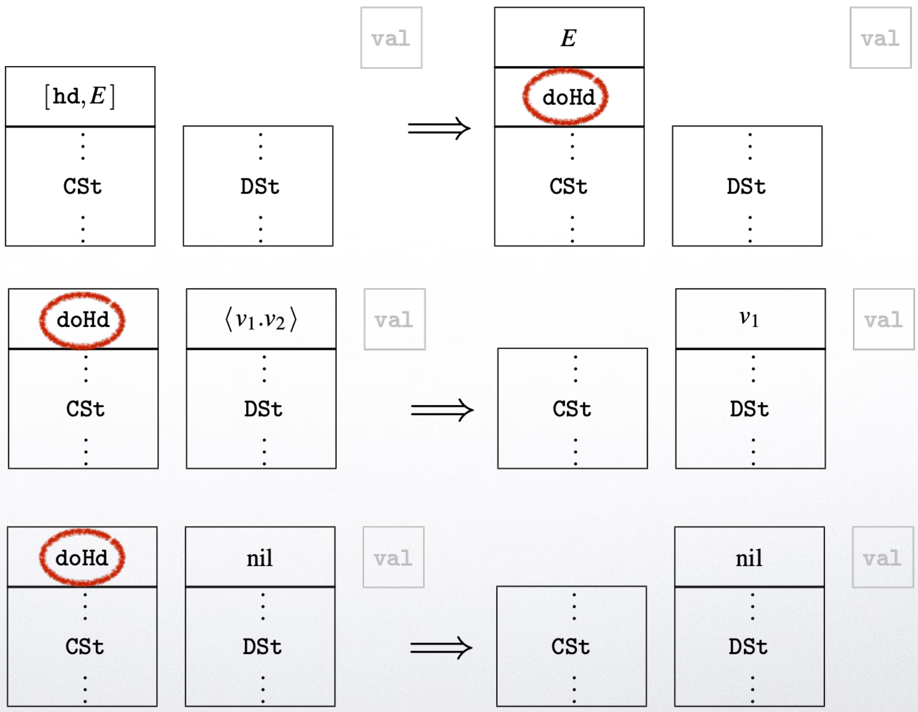
Variable



Compound Expressions

(unary and binary)

hd



(similarly for `tl`)

additional atoms used as mnemonic markers: representing what needs to be done when this marker is popped off the stack

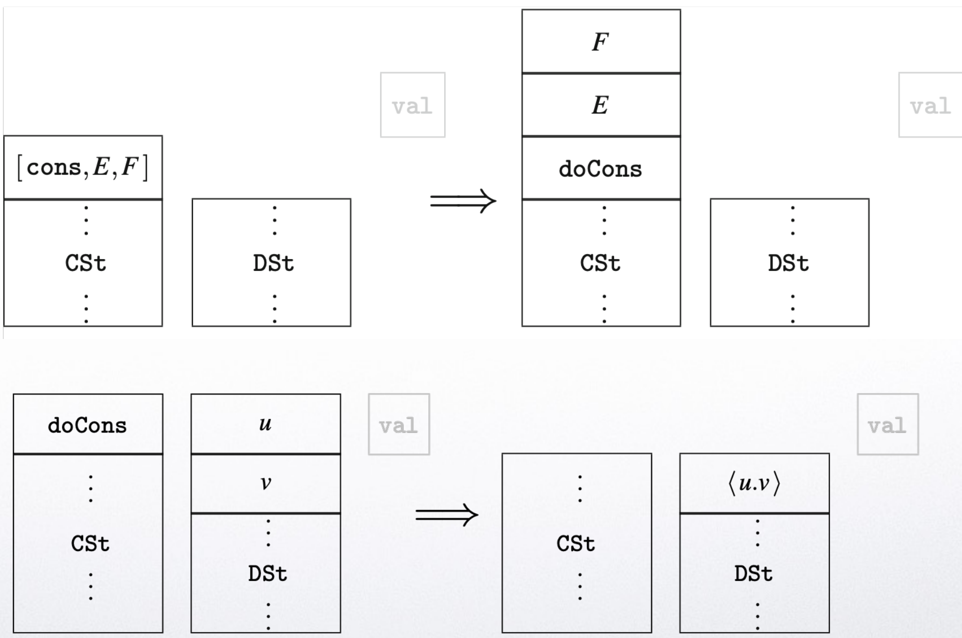
Auxiliary atoms

We now add new (encoded) atoms to $\mathbb{D}$

`doHd, doTl, doCons, doAsgn, doIf, doWhile`

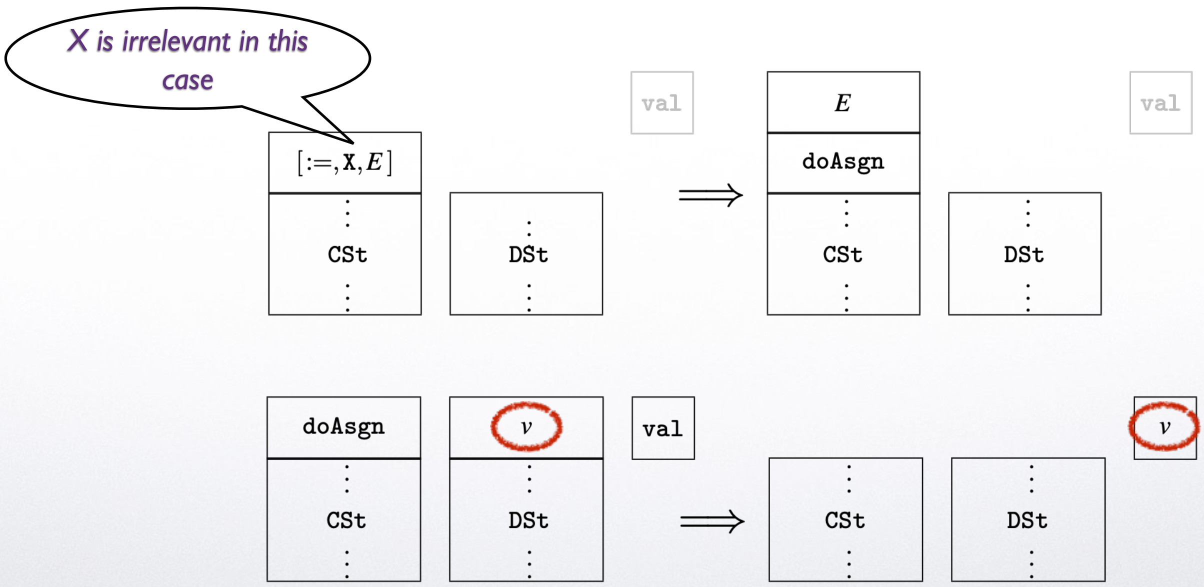
Use: push on stack to indicate operation still to be do-ne

cons

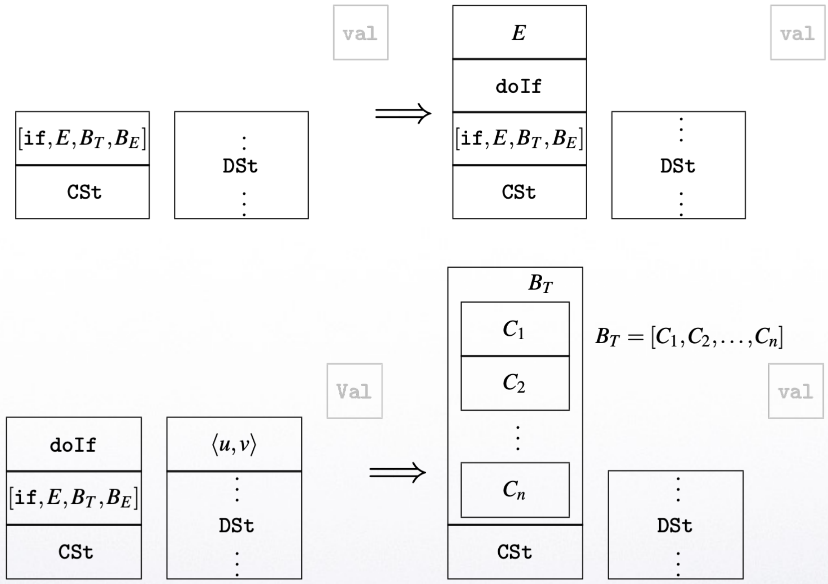


Commands

Assignment

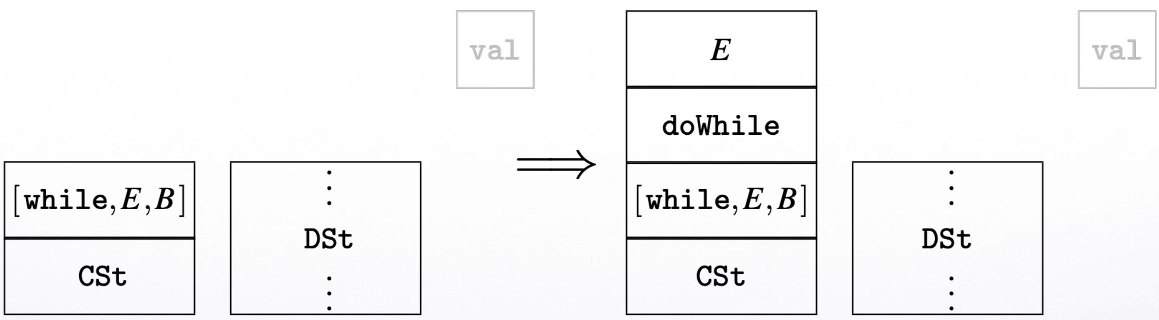


if

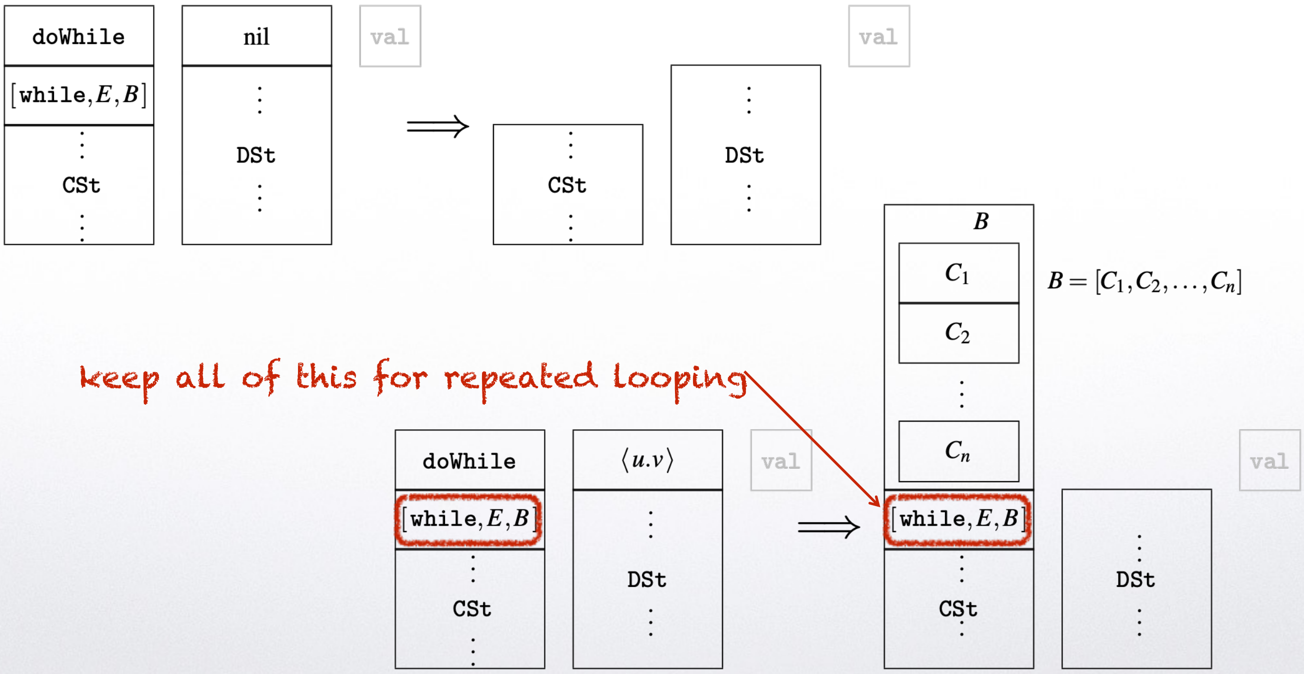


Analogously, if top element of DSt is *nil*, B_E is pushed on CSt

while

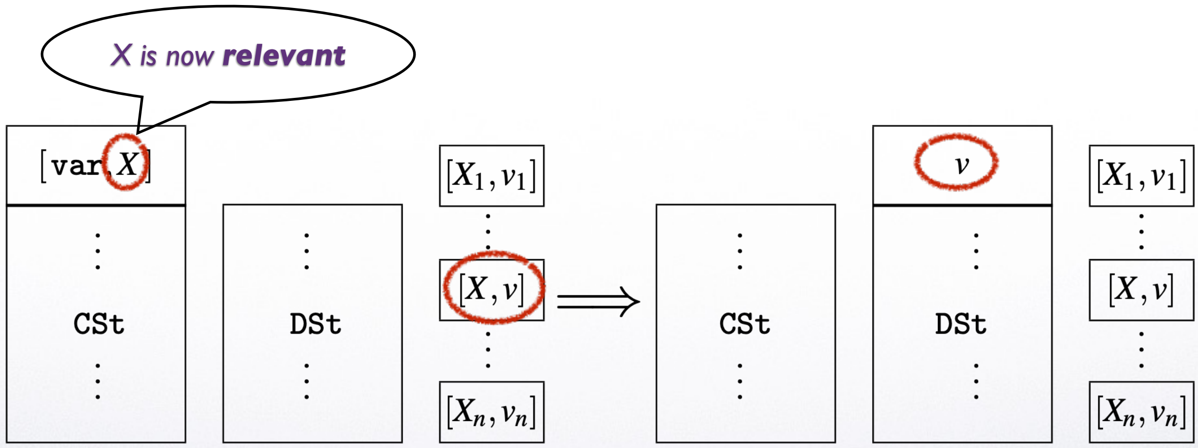


doWhile

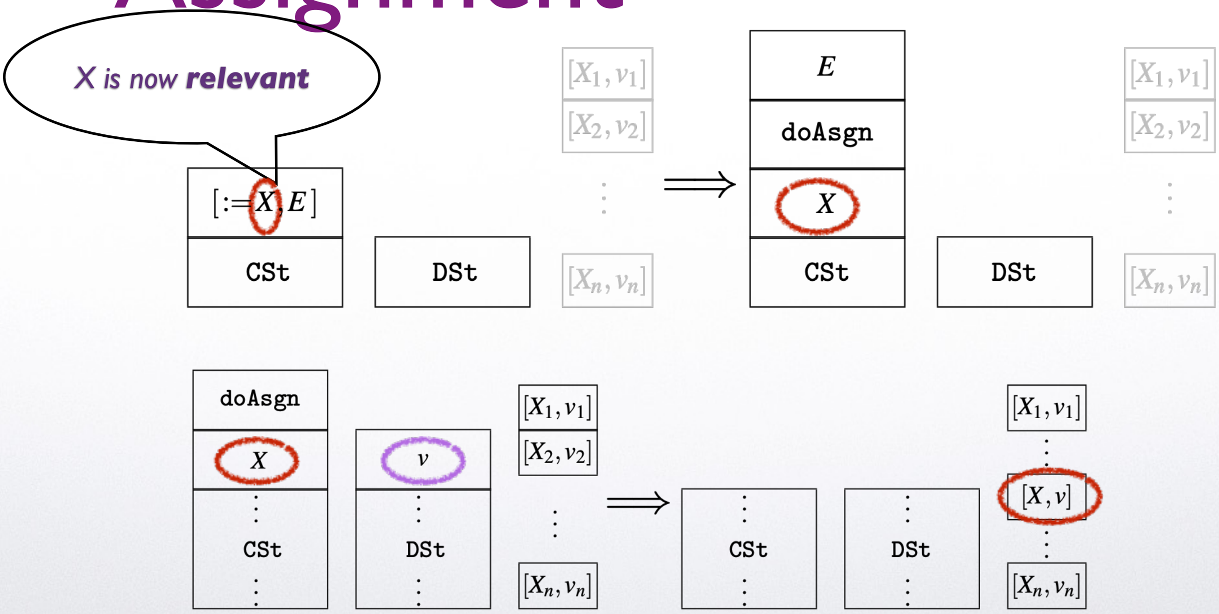


Changes to interpret WHILE

Variable lookup



Assignment



WHILE-interpreter

```
read PD {                      (* in
  P := hd PD ;
  D := hd tl PD;
  X := hd P;                               var name *)
  Y :=                                 output var name *)
                               (* B is program code block *)
                               (* CSt is code stack  *)
                               (* initially contains only B *)
  D     nil;                   (* DSt is data stack for *)
                               (* intermediate results *)
  bind := [ X, D ];
  St := [ bind ];              (* initialise store *)
  state := [ CSt, DSt, St ];   (* wrap state for STEP macro *)
  while CSt {                  (* main loop for interpretation *)
    state := <STEPn> state;    (* loop body macro  *)
    CSt := hd state            (* get command stack *)
    };
  St := hd tl tl state;        (* get final store *)
  arg := [ Y, St ];            (* wrap argument for lookup *)
  Out := <lookup> arg          (* lookup output variable value *)
 }
write Out                      (* return value of result variable *)
```

this part is as before but we now keep X and Y

CSt is code stack (code in list format),
DSt is Stack of intermediate values.
St is the the list of variable bindings

Coding the macros

- The `update` and `lookup` macro are available from Canvas, as is the main interpreter loop and the `STEPn` macro (which will be released after exercise below is completed).
- The `STEP` macro for `WH`^{1}`LE` we will complete in the exercises.

END

Next time: Our first non-computable problem