



Limits of Computation

3 - The WHILE-language
Bernhard Reus



Last time

- we discussed what problems are
- discussed that our first objective is to show that at least one of those problems cannot be “computed”
- defined what computable means in terms of “effective procedures”
- but did not commit to any specific kind of “effective procedures”



WHILE-programs as Effective Procedures

THIS TIME

- in this lecture we define a particular version of “effective procedure”:
WHILE-programs
- and how we use
WHILE’s data type

```
program read X {  
  Y := nil;           (* initial  
  while X {           (* run t  
    Y := cons hd X Y; (* appen  
    X := tl X         (* remov  
  }  
}  
write Y
```

a WHILE-program



WHILE

- Identify: ‘effective procedure’ = WHILE-program
- “*The WHILE language has just the right mix of expressive power and simplicity.*” [N. Jones]
- WHILE-programs can be interpreted on any sufficiently rich machine model...
- ...but, just like Alan Turing once did, we can define how to interpret WHILE-programs on paper (next time).
- Later we will use an interpreter.



WHILE

- WHILE-programs will be much more easily understandable, and easier to write as well, than Turing machine programs (or RAM / MIPS machine programs) which we will see much later in the term.
- The idea is that this allows you to relate the concepts presented here to your perspective as programmers (and Computer Science students).

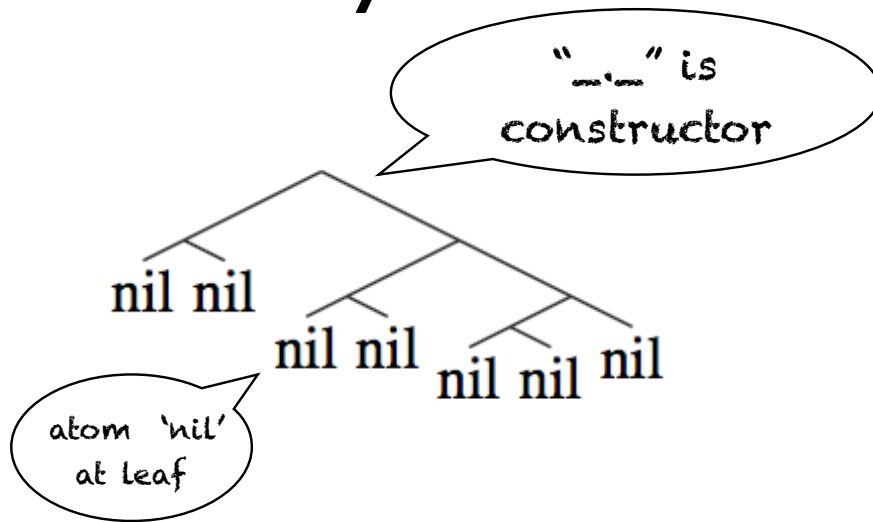


Data type: binary tree

- Our WHILE-language is *untyped*.
- Our WHILE-language has binary trees as only built-in datatype.
- allowing us to easily encode other data, including programs (!), as data values
- similar to LISP trees (or lists in other functional languages!)



Binary Trees



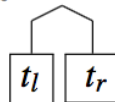
Binary Trees formally

Definition 3.1. The set of binary trees is given inductively. It contains

1. the *empty tree*:

nil

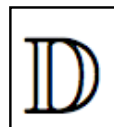
2. any tree constructed from two binary trees t_l and t_r :



and which is written $\langle t_l.t_r \rangle$ in textual notation.

3. and no other trees.

The set of binary trees is denoted \mathbb{D} (short for “data”).





Other data types?

- We can encode easily other types, for instance,
 - booleans
 - natural numbers
 - lists
- How?



Data in List Form

```
(scientist
  (id "ATM")
  (firstName "Alan")
  (midInitial "M")
  (lastName "Turing")
  (famousFor
    (achievement "crack Enigma code")
    (achievement "define computability")
  )
)
```

JSON

```
<scientist id="ATM">
  <firstName>"Alan"</firstName>
  <midInitial>"M"</midInitial>
  <lastName>"Turing"</lastName>
  <famousFor>
    <achievement>"crack Enigma code"</achievement>
    <achievement>"define computability"</achievement>
  </famousFor>
</scientist>
```

LISP S-expressions

```
{
  "scientist": {
    "id": "ATM",
    "firstName": "Alan",
    "midInitial": "M",
    "lastName": "Turing",
    "famousFor": {
      { "achievement" : "crack Enigma code" },
      { "achievement": "define computability" }
    }
  }
}
```

XML

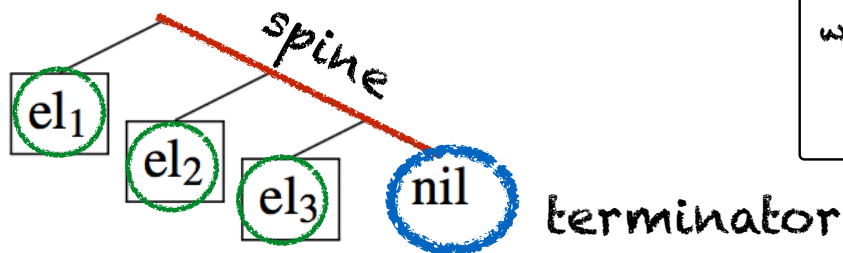


Lists

Definition 3.4. The empty list is encoded by the empty tree nil and appending an element at the front of the list is modelled by $\langle _ . _ \rangle$. More formally we define:

$$\lceil [] \rceil = \text{nil} \quad (3.1)$$

$$\lceil [a_1, a_2, \dots, a_n] \rceil = \langle \lceil a_1 \rceil . \langle \lceil a_2 \rceil . \langle \dots \langle \lceil a_n \rceil . \text{nil} \rangle \dots \rangle \rangle \quad (3.2)$$



we use $\lceil _ \rceil$ to
denote
encodings



Example

$$\lceil [[], []] \rceil = \langle \text{nil} . \langle \text{nil} . \text{nil} \rangle \rangle$$





Booleans and Numbers

Definition 3.3. We encode Boolean values as follows:

we use $\ulcorner _ \urcorner$ to
denote
encodings

$$\ulcorner \text{false} \urcorner = \text{nil}$$

$$\ulcorner \text{true} \urcorner = \langle \text{nil}.\text{nil} \rangle$$

Definition 3.5. We encode numbers inductively as follows:

$$\ulcorner 0 \urcorner = \text{nil}$$

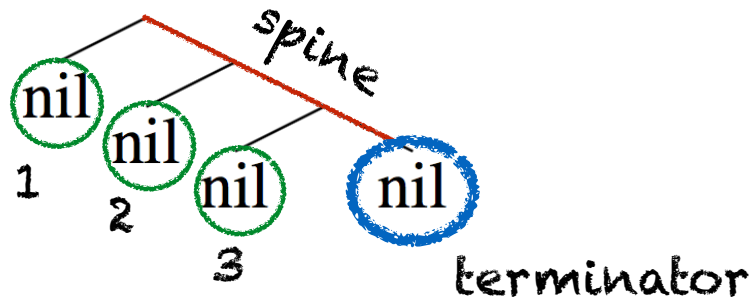
$$\ulcorner n + 1 \urcorner = \langle \text{nil}.\ulcorner n \urcorner \rangle$$



Examples

$$\ulcorner 1 \urcorner = \langle \text{nil}.\ulcorner 0 \urcorner \rangle = \langle \text{nil}.\text{nil} \rangle$$

$$\ulcorner 3 \urcorner = \langle \text{nil}.\ulcorner 2 \urcorner \rangle = \langle \text{nil}.\langle \text{nil}.\ulcorner 1 \urcorner \rangle \rangle = \langle \text{nil}.\langle \text{nil}.\langle \text{nil}.\ulcorner 0 \urcorner \rangle \rangle \rangle = \langle \text{nil}.\langle \text{nil}.\langle \text{nil}.\text{nil} \rangle \rangle \rangle$$



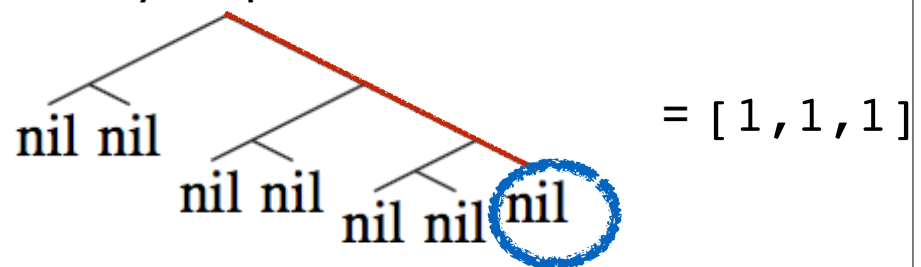


Trees as Lists



- **Any** tree can be interpreted as a list (of something). Why?

There is always a spine & terminator!



WHILE Syntax



BNF Grammar for WHILE

Expressions

```
 $\langle expression \rangle ::= \langle variable \rangle$  (variable expression)  
| nil (atom nil)  
| cons  $\langle expression \rangle$   $\langle expression \rangle$  (construct tree)  
| hd  $\langle expression \rangle$  (left subtree)  
| tl  $\langle expression \rangle$  (right subtree)  
| (  $\langle expression \rangle$  ) (right subtree)
```

Statement (Lists)

```
 $\langle block \rangle ::= \{ \langle statement-list \rangle \}$  (block of commands)  
|  $\{ \}$  (empty block)  
  
 $\langle statement-list \rangle ::= \langle command \rangle$  (single command list)  
|  $\langle command \rangle ; \langle statement-list \rangle$  (list of commands)  
  
 $\langle elseblock \rangle ::= \text{else } \langle block \rangle$  (else-case)  
  
 $\langle command \rangle ::= \langle variable \rangle := \langle expression \rangle$  (assignment)  
| while  $\langle expression \rangle$   $\langle block \rangle$  (while loop)  
| if  $\langle expression \rangle$   $\langle block \rangle$  (if-then)  
| if  $\langle expression \rangle$   $\langle block \rangle$   $\langle elseblock \rangle$  (if-then-else)
```

Programs

```
 $\langle program \rangle ::= \langle name \rangle$  read  $\langle variable \rangle$   
|  $\langle block \rangle$   
| write  $\langle variable \rangle$ 
```



BNF: Expressions

```
 $\langle expression \rangle ::= \langle variable \rangle$  (variable expression)  
| nil (atom nil)  
| cons  $\langle expression \rangle$   $\langle expression \rangle$  (construct tree)  
| hd  $\langle expression \rangle$  (left subtree)  
| tl  $\langle expression \rangle$  (right subtree)  
| (  $\langle expression \rangle$  ) (right subtree)
```


$$\langle \textit{elseblock} \rangle \quad := \textbf{else } \langle \textit{block} \rangle \quad (\text{else-case})$$

$\langle command \rangle$	$::= \langle variable \rangle := \langle expression \rangle$	(assignment)
	while $\langle expression \rangle$ $\langle block \rangle$	(while loop)
	if $\langle expression \rangle$ $\langle block \rangle$	(if-then)
	if $\langle expression \rangle$ $\langle block \rangle$ $\langle elseblock \rangle$	(if-then-else)



one output

this is where the
magic happens



END

© 2008-20. Bernhard Reus, University of Sussex

Next time:
WHILE semantics
and extensions