Limits of Computation

5- Extensions of the `WHILE` language

Bernhard Reus

Last time

- we introduced `WHILE`,
- a simple imperative untyped language,
- which has a built-in data type of binary trees (lists)
- that can be used to encode other data types.
- And discussed its semantics in Lecture 4.

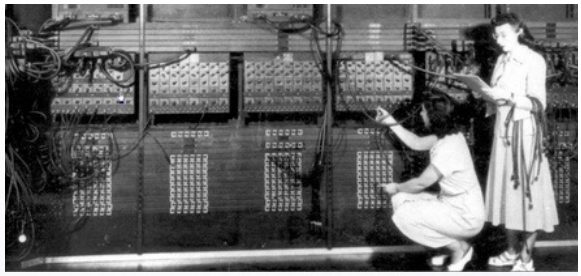
Limits of Computation

5- Extensions of the `WHILE` language

Bernhard Reus

Programming Convenience

- we present some extensions to the core `WHILE`-language
- for convenience and ease of programming
- these extensions **do not require** additional semantics, they are "syntax sugar"
- i.e. they can be translated (away) into core `WHILE`



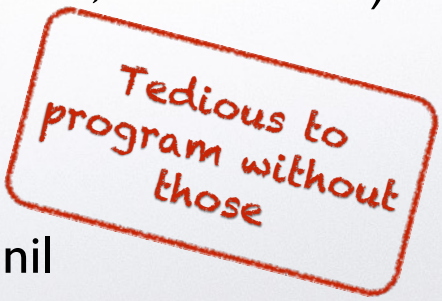
Marlyn Wescoff, standing, and Ruth Lichterman reprogram the ENIAC in 1946.

In the early days of computers, "programming" meant "plugging", not very convenient!

www.quora.com/How-was-the-very-first-programming-software-made

Core WHILE

- no built-in equality (only test for nil)
- no procedures or macros
- no number literals (nor Boolean literals, tree literals)
- no built in list notation
- no case/switch statement
- only one "atom" at leaves of trees: nil



Equality

- Equality needs to be programmed (exercises) in core WHILE
- Extended WHILE uses a new expression:

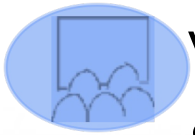
= is in *infix* notation

$$\langle expression \rangle ::= \ldots \mid \langle expression \rangle = \langle expression \rangle \mid \vdots$$

e.g. `if X=Y { Z:= X } else {Z:= Y}`

Associativity of equality

```
eqtest read X {
if X = 2 = 1 {
  Y := 1
 }
}
write Y
```



What is the output?

- For input X = 2
- For input X = 0

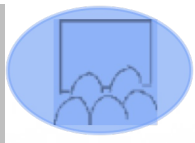
Answer: Depends on what X = 2 = 1 evaluates to!

(X = 2) = 1

X = (2 = 1)

Associativity of equality

```
eqtest read X {
if X = 2 = 1 {
  Y := 1
 }
}
write Y
```



What is the output?

- For input X = 2
- For input X = 0

Use parenthesis to ensure readability

Left (to right) associative

(X = 2) = 1

(2=2)=1 evaluates to true

(0=2)=1 evaluates to false

Right (to left) associative

X = (2 = 1)

2=(2=1) evaluates to false

0=(2=1) evaluates to true

Literals

- literals abbreviate constant values
- in our case: *natural* numbers or *Boolean* values
- Extended `WHILE` uses new expressions:

$$\langle expression \rangle ::= \ldots \mid \langle number \rangle \mid \vdots$$

$$\langle expression \rangle ::= \ldots \mid \texttt{true} \mid \texttt{false} \mid \vdots$$

e.g. `if true { Z:= cons 3 cons 1 nil }`

Lists

- lists can be encoded in our datatype but explicit syntax is nicer
- Extended `WHILE` uses new expressions:

$$\begin{array}{lll} \langle expression \rangle & ::= \ldots & \\ & \mid [\,] & \text{(empty list constructor)} \\ & \mid [\,\langle expression\text{-}list \rangle\,] & \text{(nonempty list constructor)} \\ & \vdots & \\ \langle expression\text{-}list \rangle & ::= \langle expression \rangle & \text{(single expression list)} \\ & \mid \langle expression \rangle\,,\,\langle expression\text{-}list \rangle & \text{(multiple expression list)} \end{array}$$

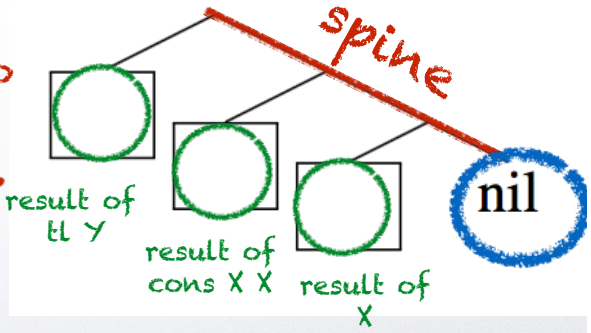
List Example

```
[tl Y, cons X X, X]
```

translates to

```
cons (tl Y)
  (cons (cons X X)
        cons X nil)
```

evaluates to



Macro Calls

- We don't have procedures but we can implement "macro calls" that allows one:
  - to write more readable code
  - to write modular code as macro code can be replaced without having to change the program.

Procedures provide abstraction & modularisation

Syntax of Macro Calls

- Macro calls use angle brackets <…> around the name of the program called
- and one argument (programs have one argument)
- Extended `WHILE` uses new assignment command:

$$\langle command \rangle ::= \ldots$$
$$\quad | \quad \langle variable \rangle := < \langle name \rangle > \langle expression \rangle$$
$$\quad \vdots$$

Macro Calls Example

```
succ read X {
   X := cons nil X
}
write X
```

```
pred read X {
   X := tl X
}
write X
```

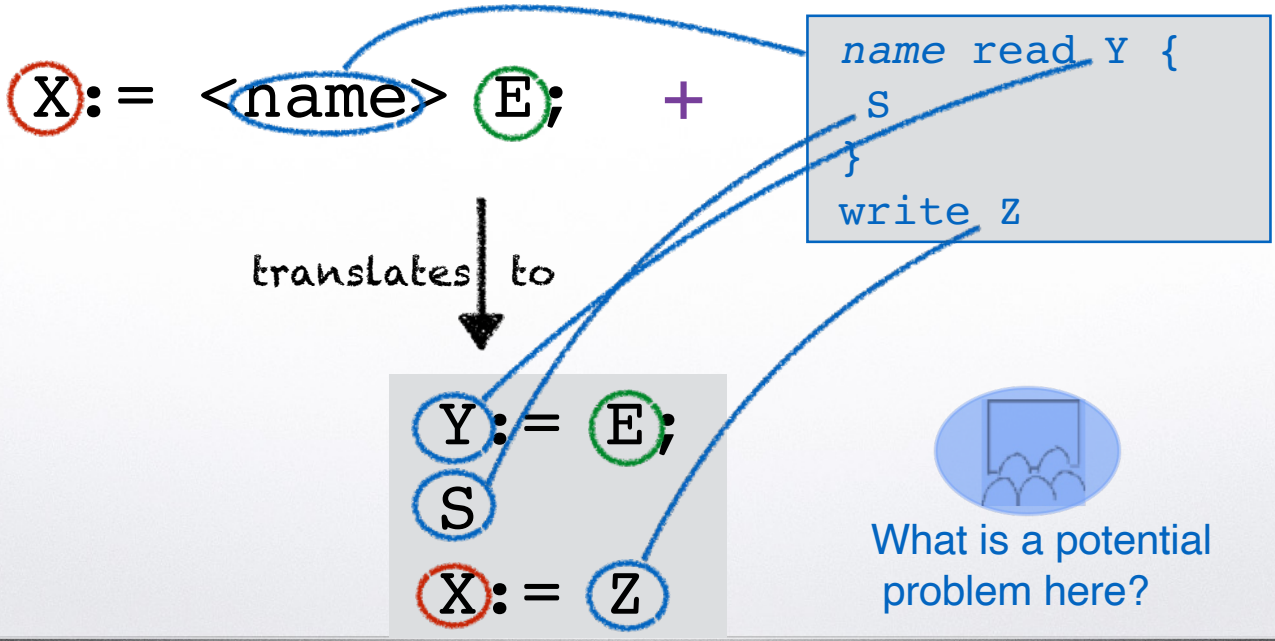
```
add read L {
  X:= hd L;
  Y:= hd tl L;
  while X {
        X := <pred> X;
        Y := <succ> Y
        }
}
write Y
```

Semantics of Macro Calls

X := <name> E

1. Evaluate argument E (expression) to obtain *d*
2. Run the body of macro name with input *d*
3. And obtain as result r
4. Assign the value r to variable X

Translate Macros Calls



Switch Statement

luxurious form of if-then-else cascade

⟨*command*⟩ ::= …
| **switch** ⟨*expression*⟩ { ⟨*rule-list*⟩ }
| **switch** ⟨*expression*⟩ { ⟨*rule-list*⟩
default : ⟨*statement-list*⟩ }
⋮

⟨*rule*⟩ ::= **case** ⟨*expression-list*⟩ : ⟨*statement-list*⟩

⟨*rule-list*⟩ ::= ⟨*rule*⟩
| ⟨*rule*⟩ ⟨*rule-list*⟩

Switch Example

```
switch X  {
 case 0          :  Y := 0
 case 1, 3       :  Y := 1
 case cons 2 nil :  Y := 2
 }
```

evaluates to [2]

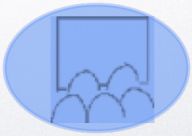
translates to

```
if X = 0
  { Y := 0 }
else { if X = 1
       { Y := 1 }
     else
       { if X = 3
           { Y := 1 }
         else
           {  if X = cons 2 nil
                { Y := 2 }
           }
       }
     }
```

Extra Atoms

- Atoms are the "labels" at the leaves of binary trees.
- So far only one atom: nil
- Add more to simplify encodings.
- Extended `WHILE` uses new expression(s):

$$\langle expression \rangle ::= \ldots \mid \texttt{a} \quad (a \in Atoms) \mid \vdots$$



Only finitely many. Why?

Tree Literals

- literals abbreviate constant values
- What about trees?
- Extended `WHILE` uses new expressions:

$$\langle expression \rangle ::= \ldots \mid \langle\ tree\ \rangle \mid \vdots$$

trees in $\mathbb{D}$

e.g. `if X { Z:= <nil.nil> }`

END

Next time:
`WHILE`-programs as
`WHILE`-data