



Limits of Computation

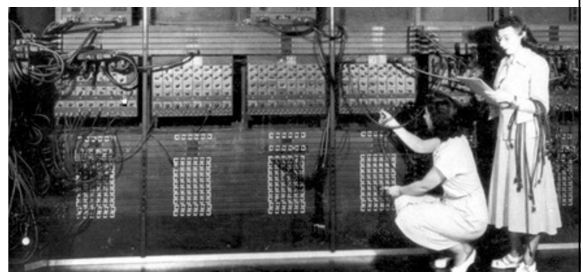
5- Extensions of the WHILE language Bernhard Reus

2



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

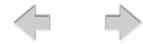


Marlyn Wescoff, standing, and Ruth Lichterman reprogram the ENIAC in 1946.

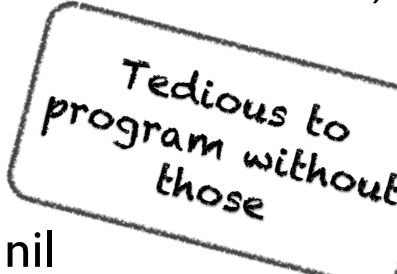
In the early days of computers, “programming” meant “plugging”, not very convenient!

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

8



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
- 
- Tedious to
program without
those

Tedious to
program without
those

9



Equality

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

$$\langle expression \rangle ::= \dots$$

= is in *infix* notation

$$|\langle expression \rangle = \langle expression \rangle$$

-
-
-

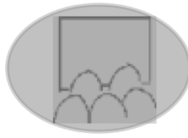
e.g. `if X=Y { Z := X } else {Z := Y}`

11



Associativity of equality

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



What is the output?

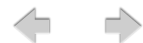
- For input $X = 2$
- For input $X = 0$

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

$$(X = 2) = 1$$

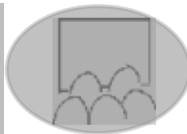
$$X = (2 = 1)$$

12



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

13



Literals

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

$\langle expression \rangle ::= \dots$	$\langle expression \rangle ::= \dots$
$\langle number \rangle$	true
\vdots	false
	\vdots

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

14



Lists

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

$\langle expression \rangle ::= \dots$	
<code>[]</code>	(empty list constructor)
<code>[$\langle expression-list \rangle$]</code>	(nonempty list constructor)
\vdots	
$\langle expression-list \rangle ::= \langle expression \rangle$	(single expression list)
<code>$\langle expression \rangle$, $\langle expression-list \rangle$</code>	(multiple expression list)

15



List Example

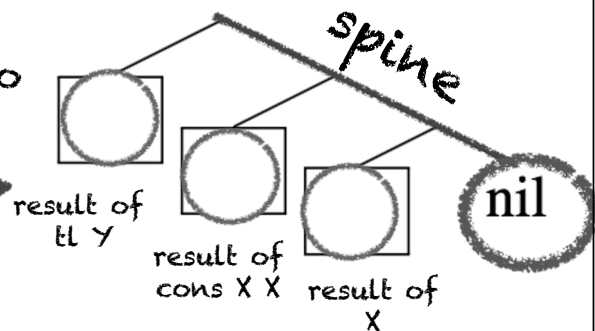
`[tl Y, cons X X, X]`

translates to

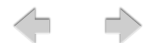


```
cons (tl Y)
  (cons (cons X X)
        cons X nil)
```

evaluates to



16

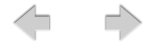


Macro Calls

- 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

22



Syntax of Macro Calls

- Macro calls use angle brackets $\langle \dots \rangle$ around the name of the program called
- and one argument (programs have one argument)
- Extended WHILE uses new assignment command:

$\langle \text{command} \rangle ::= \dots$
 | $\langle \text{variable} \rangle := \langle \text{name} \rangle \langle \text{expression} \rangle$
 | \vdots

23



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 :=  $\langle \text{pred} \rangle$  X;  
    Y :=  $\langle \text{succ} \rangle$  Y  
  }  
}  
write Y
```

24



Semantics of Macro Calls

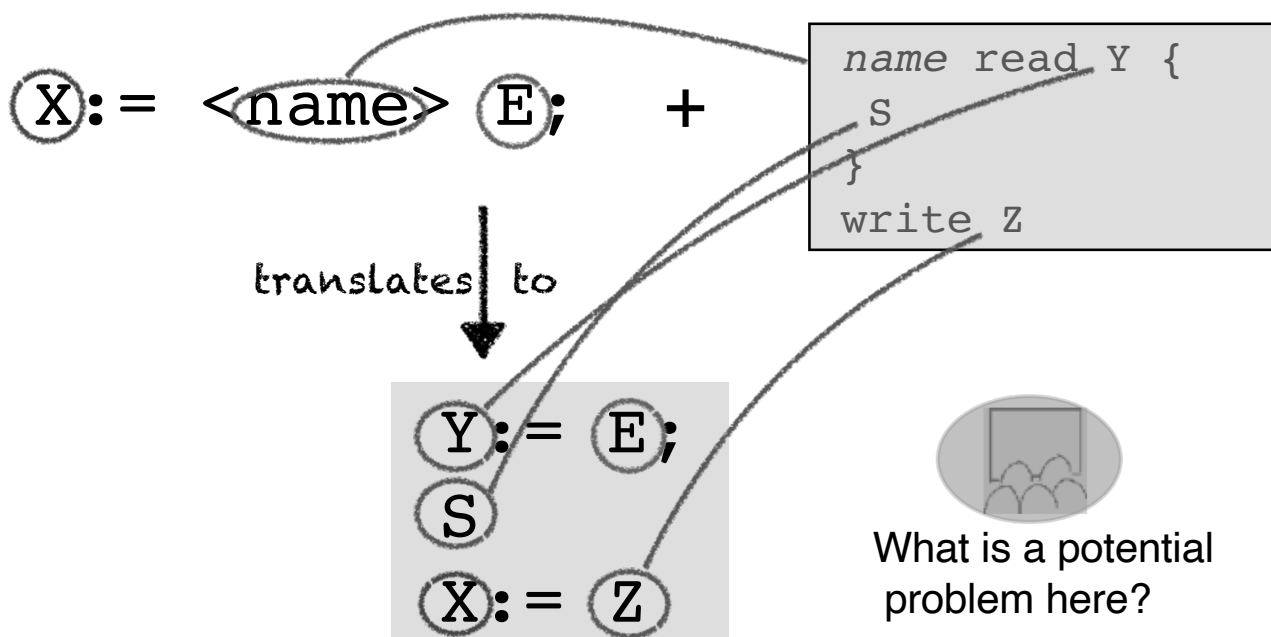
$X := \langle \text{name} \rangle E$

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

25



Translate Macros Calls



27



Switch Statement

luxurious form of if-then-else cascade

```
 $\langle command \rangle ::= \dots$   
| switch  $\langle expression \rangle$  {  $\langle rule-list \rangle$  }  
| switch  $\langle expression \rangle$  {  $\langle rule-list \rangle$  }  
| default:  $\langle statement-list \rangle$   
|  
|  
|  
| $\vdots$ 
```

```
 $\langle rule \rangle ::= \text{case } \langle expression-list \rangle : \langle statement-list \rangle$ 
```

```
 $\langle rule-list \rangle ::= \langle rule \rangle$   
|  $\langle rule \rangle \langle rule-list \rangle$ 
```

28



Switch Example

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

evaluates to [2] translates to

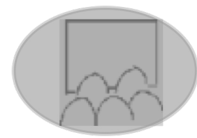
```
if X = 0  
  { Y := 0 }  
else { if X = 1  
      { Y := 1 }  
      else  
        { if X = 3  
          { Y := 1 }  
          else  
            { if X = cons 2 nil  
              { Y := 2 }  
            }  
          }  
        }  
}
```

29



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):

$$\begin{array}{l} \langle expression \rangle ::= \dots \\ \quad | \quad a \quad (a \in Atoms) \\ \quad \vdots \end{array}$$


Only finitely many.
Why?

30



END

© 2008-24. Bernhard Reus, University of Sussex

Next time:
WHILE-programs as
WHILE-data

31