Object-oriented programming without built-in types

Software Engineering, Theory and Experimental Programming (STEP-2024)

Computing the Answer to the Ultimate Question to Life, the Universe, and Everything

Alexander Kogtenkov

03

2024–04–~~01~~

Disclaimer

No affiliation

No use

No science

Motivation

Effective computability ("Propositions as types" by Philip Wadler):

- Alonzo Church: Lambda calculus
- Kurt Gödel: Recursive functions
- Alan M. Turing: Turing machines

Motivation

Effective computability:

- Alonzo Church: Lambda calculus

$\mathbf{0} := \lambda f.\lambda x.x$

$\mathbf{1} := \lambda f.\lambda x.f\ x$ $\qquad$ $\mathbf{plus} := \lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x)$

$\mathbf{2} := \lambda f.\lambda x.f\ (f\ x)$ $\qquad$ $\mathbf{mult} := \lambda m.\lambda n.\lambda f.m\ (n\ f)$

$\ldots$

Motivation

Effective computability:

- Alonzo Church: Lambda calculus

$$\textbf{true} := \lambda x.\lambda y.x \qquad \textbf{and} := \lambda p.\lambda q.p\ q\ p$$

$$\textbf{false} := \lambda x.\lambda y.y \qquad \textbf{if\_then\_else} := \lambda p.\lambda a.\lambda b.p\ a\ b$$

Motivation

Effective computability:

- Alonzo Church: Lambda calculus

  1. Variables: x, y, ...
  2. Abstraction: λx.expr
  3. Application: foo bar

Effective computability:

- Alonzo Church: Lambda calculus

    1. Variables: x, y
    2. Ab .expr
    3. pplication: foo bar

Can we do similarly in **pure** OOP?

Motivation

Bonus: The Secret Notation for Functional Programming in Eiffel

https://youtu.be/zxBYH9nrykI

Effective computability:

- Alonzo Church: Lambda calculus

  1. Variables: x, y
  2. Ab… x.expr
  3. …pplication: foo bar

Can we do similarly in **pure** OOP?

Motivation

Model languages in formal proofs

Alexander J. Summers and Peter Müller. "Freedom Before Commitment: A Lightweight Type System for Object Initialisation". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. Portland, Oregon, USA: ACM, 2011, pp. 1013–1032. ISBN: 978-1-4503-0940-0. DOI: 10.1145/2048066.2048142

Classes:

- parent?
- field*
- method*

Expressions:

$$e ::= x \mid x.f \mid \textbf{null}$$

Instructions:

$$\begin{aligned} s ::= \quad & x = e \\ \mid \quad & z.f = y \\ \mid \quad & x = y.m\ (\bar{z}) \\ \mid \quad & x = \textbf{new}\ C\ (\bar{z}) \\ \mid \quad & x = (t)\ y \\ \mid \quad & s_1\ ;\ s_2 \end{aligned}$$

Motivation

Model languages in formal proofs

Alexander J. Summers and Peter Müller. "Freedom Before Commitment: A Lightweight Type System for Object Initialisation". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. Portland, Oregon, USA: ACM, 2011, pp. 1013–1032. ISBN: 978-1-4503-0940-0. DOI: 10.1145/2048066.2048142

Classes:

- parent?
- field*
- method*

Expre

$$e ::= x \mid x.f \mid \textbf{null}$$

Instructions:

$$s ::= x = e$$

$$\mid \quad x = y.m\ (\bar{z})$$

$$\mid \quad x = \textbf{new}\ C\ (\bar{z})$$

$$\mid \quad x = (t)\ y$$

$$\mid \quad s_1\ ;\ s_2$$

Can we do **anything useful** with it?

Model languages in formal proofs

Alexander J. Summers and Peter Müller. "Freedom Before Commitment: A Lightweight Type System for Object Initialisation". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. Portland, Oregon, USA: ACM, 2011, pp. 1013–1032. ISBN: 978-1-4503-0940-0. DOI: 10.1145/2048066.2048142

Classes:

The Answer to the Question

- field*
- method*

Expre

$e ::= x \mid x.f \mid \textbf{null}$

Instructions:

$s ::= x = e$

$\mid x = y.m\ (\bar{z})$

$\mid x =$ new $C\ (\bar{z})$

$\mid x = (t)\ y$

$\mid s_1\ ;\ s_2$

Side effects!

Can we do **anything useful** with it?

Model language

Included

Classes:

parent*

method*

field*

Expressions:

$x \mid f \mid x.m\ (args)$

$\mid$ **Result** $\mid$ **Current**

Instructions:

$x := e$

$f := e$

create $x.m\ (args)$

$x.m\ (args)$

$i_1;\ i_2$

Not included

Built-in types:

Integer

Boolean

Array

Operators:

Arithmetic

Comparison (including equality)

Branching constructs:

Conditional instructions

Loops

Multi-branch

Proof method

Proof by Necessity It had better be true or the whole structure of mathematics would crumble to the ground.

Proof by Lack of Sufficient Time Because of the time constraint, I'll leave the proof to you.

Proof by Margin Size This theorem has a truly marvelous proof which this margin is too narrow to contain.

Proof by Calculus This proof requires calculus, so we'll skip it.

Proof by Tessellation This proof is just the same as the last.

Proof by Accumulated Evidence Long and diligent search has not revealed a counterexample.

Proof by Deferral We'll prove this later in the seminar.

Proof by Intimidation Trivial.

Proof method

Proof by Necessity It had better be true or the whole structure of mathematics would crumble to the ground.

Proof by Lack of Sufficient Time Because of the time constraint, I'll leave the proof to you.

Proof by Margin Size This theorem has a t[illegible] [illegible]ch this margin is too narrow to contai[illegible]

Proof by Calculus T[illegible] [illegible]ulus, so we'll skip it.

Proof by [illegible] [illegible]oof is just the same as the last.

Proof by [illegible]ulated **Evidence** Long and diligent search has not revealed a counterexample.

Proof by Deferral We'll prove this later in the seminar.

Proof by Intimidation Trivial.

Proof by Example I'll show you an example and the remaining cases you can prove yourself.

The question

Ultimate Question to Life, the Universe, and Everything

1979 UK, Douglas Adams

The question

Ultimate Question to Life, the Universe, and Everything

1979 UK, Douglas Adams

Most popular random number between 1 and 100

5 days ago US, Veritasium

The question

Ultimate Question to Life, the Universe, and Everything

1979 UK, Douglas Adams

Most popular random number between 1 and 100

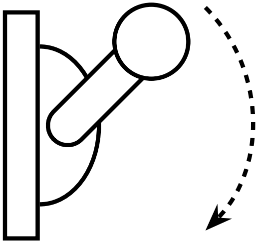
5 days ago US, Veritasium

Fibonacci numbers

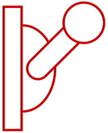
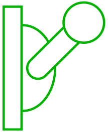
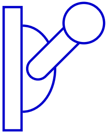
450–200BC India; 1202 Pisa, Fibonacci

How to achieve without passing data?

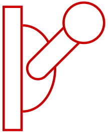
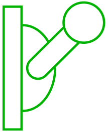
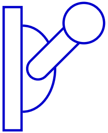
Side effects



Side effects

Trigger	API
	*reset*
	*increment*
	*emit*

Side effects

Trigger	API	External behavior in C
	*reset*	`byte = 0;`
	*increment*	`byte++;`
	*emit*	`putc (byte);`

Side effects

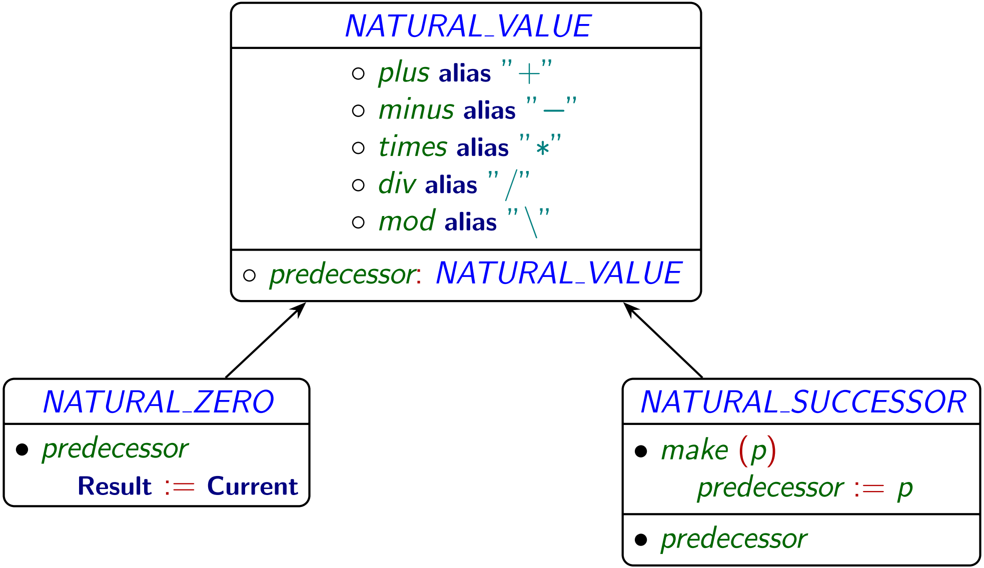
$$\textbf{create}\ \mathit{output}.\mathit{make}$$

$$\mathit{output}.\mathit{reset}$$

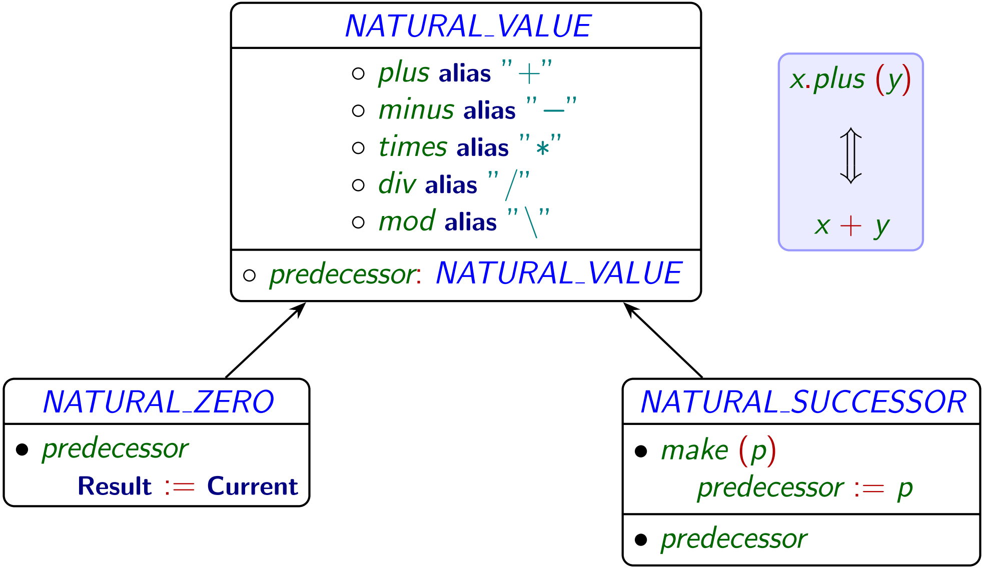
$$\left.\begin{array}{l} \mathit{output}.\mathit{increment} \\ \dots \\ \mathit{output}.\mathit{increment} \end{array}\right\} 42\text{times}$$

$$\mathit{output}.\mathit{emit}\ \text{——Prints '}\ast\text{'}$$

Natural numbers



Natural numbers



Addition

plus **alias** "+" (*other*: *NATURAL_VALUE*): *NATURAL_VALUE*

$\mathbf{0} + \mathsf{n} = \mathsf{n}$

NATURAL_ZERO:

Result := *other*

$(\mathsf{m} + 1) + \mathsf{n} = \mathsf{m} + (\mathsf{n} + 1)$

NATURAL_SUCCESSOR:

Result := *predecessor* + **create** {*NATURAL_SUCCESSOR*}.*make* (*other*)

Multiplication

times **alias** "$*$" (*other*: *NATURAL_VALUE*): *NATURAL_VALUE*

$\mathbf{0} \times \mathsf{n} = \mathbf{0}$

NATURAL_ZERO:

Result := **Current**

$(\mathsf{m} + 1) \times \mathsf{n} = \mathsf{m} \times \mathsf{n} + \mathsf{n}$

NATURAL_SUCCESSOR:

Result := *predecessor* $*$ *other* $+$ *other*

Literals

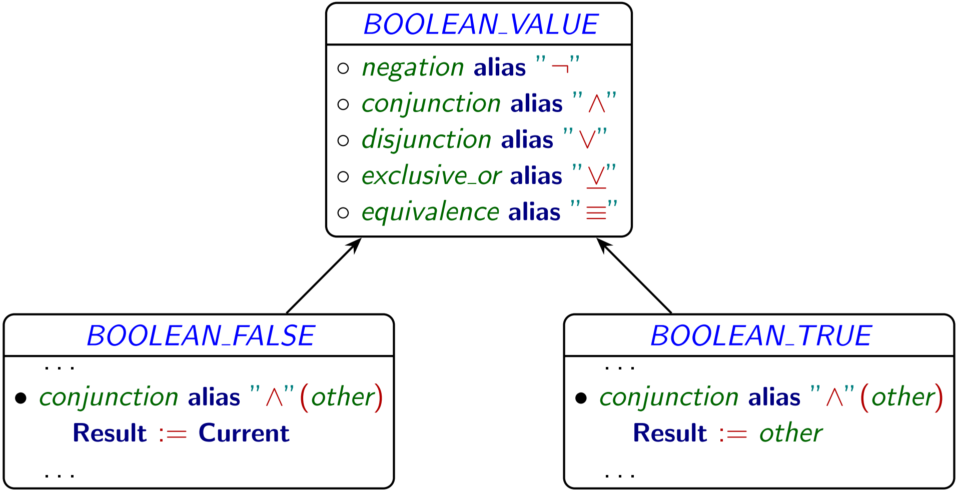
```
n0: NATURAL_VALUE do
    create {NATURAL_ZERO} Result end

n1: NATURAL_VALUE do
    create {NATURAL_SUCCESSOR} Result.make (n0) end

n2: NATURAL_VALUE do Result := n1 + n1 end
n3: NATURAL_VALUE do Result := n2 + n1 end
...
n10: NATURAL_VALUE do Result := n9 + n1 end

n20: NATURAL_VALUE do Result := n2 * n10 end
n30: NATURAL_VALUE do Result := n3 * n10 end
...
```

Literals

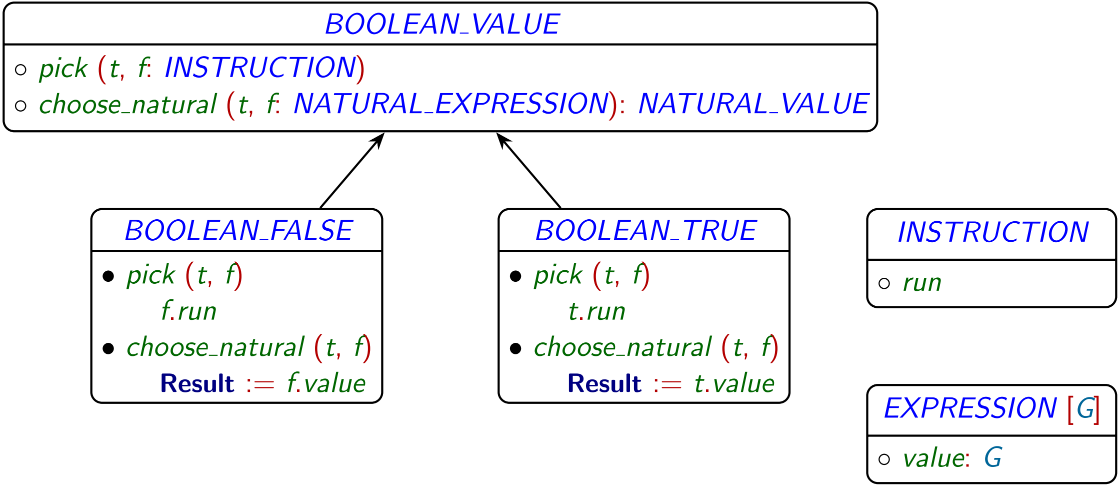
```
n0: NATURAL_VALUE do
    create {NATURAL_ZERO} Result end

n1: NATURAL_VALUE do
    create {NATURAL_SUCCESSOR} Result.make (n0) end

n2: NATURAL_VALUE do Result := n1 + n1 end
n3: NATURAL_VALUE do Result := n2 + n1 end
...
n10: NATURAL_VALUE do Result := n9 + n1 end

n20: NATURAL_VALUE do Result := n2 * n10 end
n30: NATURAL_VALUE do Result := n3 * n10 end
...
```

```
n42 := n40 + n2
```

Boolean values



Boolean values



Subtraction

```
minus alias "–" (other: NATURAL_VALUE): NATURAL_VALUE

NATURAL_ZERO:
    Result := Current

NATURAL_SUCCESSOR:
    Result := other.is_zero.choose_natural (Current, predecessor – other.predecessor)
```

$\mathbf{0} - n = \mathbf{0}$

$$\begin{aligned} (m+1) - \mathbf{0} &= m+1 \\ (m+1) - (n+1) &= m-n \end{aligned}$$

Comparison

is_less **alias** "<" $(other: NATURAL\_VALUE): BOOLEAN\_VALUE$

$\mathbf{0} < n \iff n \neq \mathbf{0}$

$NATURAL\_ZERO$:

Result $:= \neg\ other.is\_zero$

$(m + 1) < \mathbf{0} \iff \textbf{false}$

$(m + 1) < (n + 1) \iff m < n$

$NATURAL\_SUCCESSOR$:

Result $:= (\neg\ other.is\_zero) \land (predecessor < other.predecessor)$

Comparison

is_less **alias** "<" (*other*: *NATURAL_VALUE*): *BOOLEAN_VALUE*

$$
\begin{aligned}
\mathbf{2} < \mathbf{3} &\iff \\
(\mathbf{1} + 1) < (\mathbf{2} + 1) &\iff \\
\mathbf{1} < \mathbf{2} &\iff \\
(\mathbf{0} + 1) < (\mathbf{1} + 1) &\iff \\
\mathbf{0} < \mathbf{1} &\iff \\
\mathbf{1} \neq \mathbf{0} &\iff \\
\textbf{true} &
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{3} < \mathbf{2} &\iff \\
(\mathbf{2} + 1) < (\mathbf{1} + 1) &\iff \\
\mathbf{2} < \mathbf{1} &\iff \\
(\mathbf{1} + 1) < (\mathbf{0} + 1) &\iff \\
\mathbf{1} < \mathbf{0} &\iff \\
(\mathbf{0} + 1) < \mathbf{0} &\iff \\
\textbf{false} &
\end{aligned
}$$

Why do we need *INSTRUCTION* and *EXPRESSION*?

```
if condition then
    print_1
else
    print_2
end
```

$\Longrightarrow$

```
if_ (
    condition,
    print_1,
    print_2
)
```

Why do we need *INSTRUCTION* and *EXPRESSION*?

```
if condition then
    print_1
else
    print_2
end
```

Deferred execution

$\Longrightarrow$

```
if_ (
    condition,
    print_1,
    print_2
)
```

Immediate execution

Why do we need *INSTRUCTION* and *EXPRESSION*?

```
if condition then
    print_1
else
    print_2
end
```

Deferred execution

$\Longrightarrow$

```
if_ (
    condition,
    print_1,
    print_2
)
```

~~Immediate execution~~

Why do we need *INSTRUCTION* and *EXPRESSION*?

```
if condition then
    print_1
else
    print_2
end
```

Deferred execution

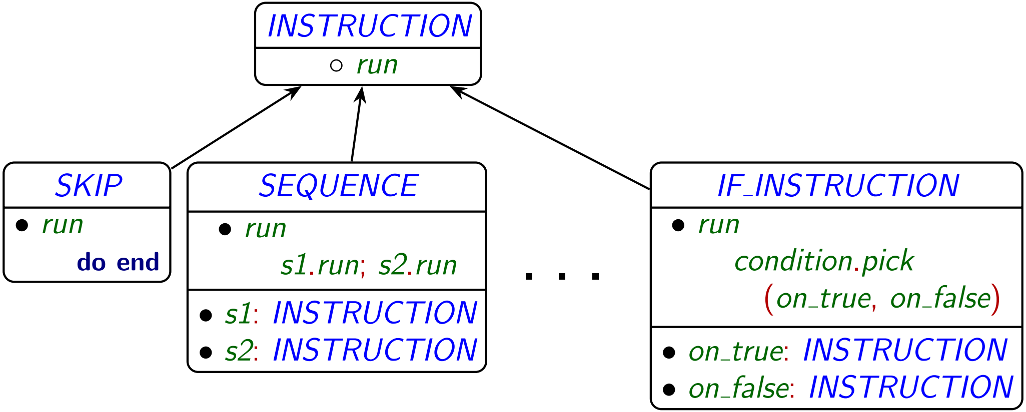
$\Longrightarrow$

```
if_ (
    condition,
    print_1,
    print_2
)
```

~~Immediate execution~~

Solution: wrap them!

Why do we need *INSTRUCTION* and *EXPRESSION*?



Assignment

$$x := expr$$

Assignment

$x := expr$

VARIABLE [*G*] **inherit** *EXPRESSION* [*G*]:

  value: *G*

  put (*v*: *G*) **do** *value* := *v* **end**

  *put_* **alias** ":=" (*e*: *EXPRESSION* [*G*]): *INSTRUCTION*

    do **create** {*ASSIGNMENT* [*G*]} **Result**.*make* (*e*, **Current**) **end**

ASSIGNMENT [*G*] **inherit** *INSTRUCTION*:

  expression: *EXPRESSION* [*G*]

  variable: *VARIABLE* [*G*]

  run **do** *variable*.*put* (*expression*.*value*) **end**

Systematic wrapping

LINKED_LIST [*G*]

Regular	Wrapped
first: *G*	*first_*: *EXPRESSION* [*G*]
item (*i*: *NATURAL_VALUE*): *G*	*item_* (*i*: *NATURAL_EXPRESSION*): *EXPRESSION* [*G*]
count: *NATURAL_VALUE*	*count_*: *NATURAL_EXPRESSION*
insert_first (*value*: *G*)	*insert_first_* (*expression*: *EXPRESSION* [*G*]): *INSTRUCTION*
remove_first	*remove_first_*: *INSTRUCTION*

Loop

until

$\boxed{\text{condition}}$

loop

$\boxed{\text{body}}$

end

Loop

BOOLEAN_EXPRESSION

until

condition

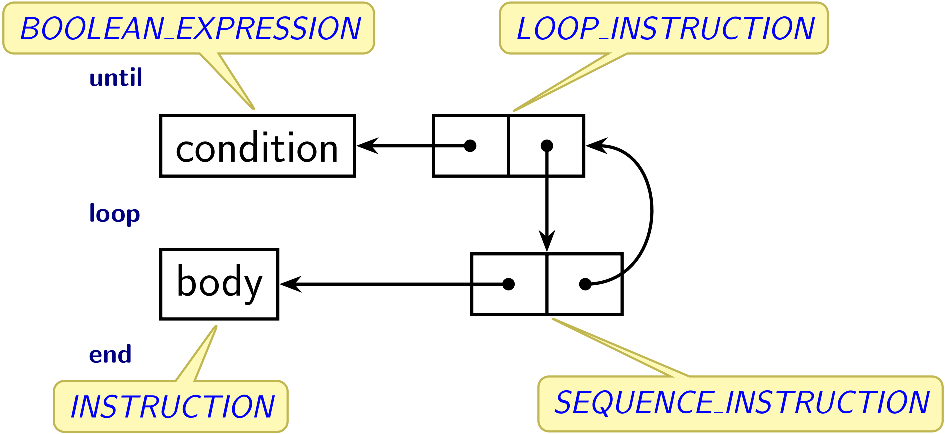
loop

body

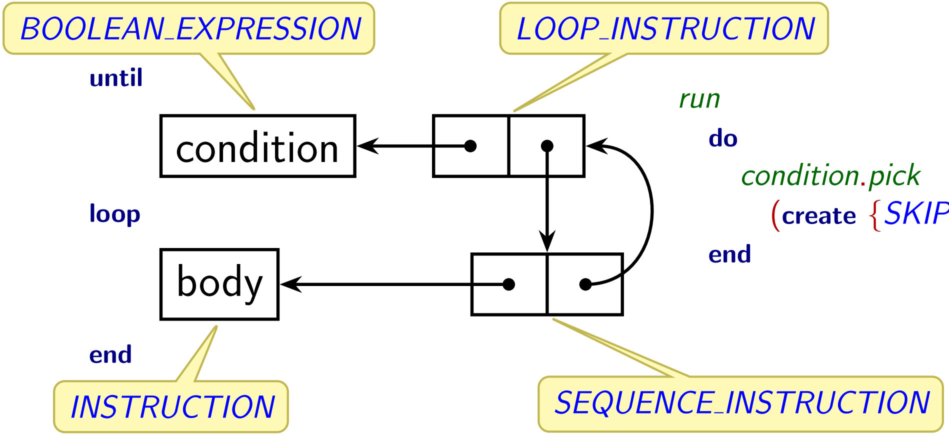
end

INSTRUCTION

Loop


BOOLEAN_EXPRESSION
LOOP_INSTRUCTION
until
condition
loop
body
end
INSTRUCTION
SEQUENCE_INSTRUCTION

Loop



run
 do
  condition.*pick*
   (**create** {*SKIP*}, *seq*)
 end

Example: item by index in *LINKED_LIST*

```
item (i: NATURAL_VALUE): G
    local
        n: LINKED_NODE_VARIABLE [G]
        j: NATURAL_VARIABLE
    do
        create n.make (head)
        create j.make (i)
        loop_until (
            j ≡ n0,
            n := n.next_ +
            j := (j − n1)
        )
        Result := n.value.item
    end
```

Model language (revised)

Included

Classes:

- parent*
- method*
- field*

Expressions:

- $x \mid f \mid x.m\ (args)$
- $\mid$ **Result** $\mid$ **Current**

Instructions:

- $x := e$
- $f := e$
- **create** $x.m\ (args)$
- $x.m\ (args)$
- $i_1;\ i_2$

Implemented

Types:

- Integer
- Boolean
- Array (as List)

Operators:

- Arithmetic
- Comparison (including equality)

Branching constructs:

- Conditional instructions
- Loops
- Multi-branch (work in progress)

More types and operations

Integer

- one natural with a sign
- two naturals with normalization

Size-limited naturals/integers (`uint8`, `int32`, etc.)

- modular arithmetic with normalization

Bit-wise operations

- powers of 2 + arithmetic

Arrays

- lists

Real numbers

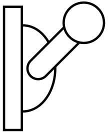
- left to audience

What about input?

Recall: no data types!

What about input?

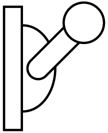
External



read

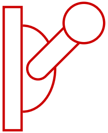
What about input?

External

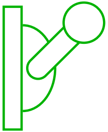


read

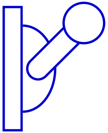
Internal



reset



increment



emit

Demo!

Summary

Key observation

Built-in types, operators, literals, branching instructions, etc. are leftovers from electrical engineers. True programmers do not need all this obsolete stuff.

What was/would be useful?

- Genericity
- Type inference
- Flexible syntax
- Fixing operator precedence in the Unicode standard

What OO language designers should be looking for?

Properties that cannot be derived from the basics of OOP

What OOPL compilers should do?

Automatically detect wrappers and generate highly efficient code instead