



Operational Semantics

One-Slide Summary

- **Operational semantics** are a precise way of specifying how to evaluate a program.
- A **formal semantics** tells you what each expression means.
- Meaning depends on **context**: a **variable environment** will map variables to memory locations and a **store** will map memory locations to values.

Lecture Outline: OpSem

- Motivation
- Notation
- **The Rules**
 - Simple Expressions
 - while
 - new
 - dispatch



Motivation

- We must specify for every Cool expression *what happens when it is evaluated*
 - This is the **meaning** of an expression
- The definition of a programming language:
 - The tokens \Rightarrow lexical analysis
 - The grammar \Rightarrow syntactic analysis
 - The typing rules \Rightarrow semantic analysis
 - The evaluation rules \Rightarrow interpretation
(also: hints for compilation)

Evaluation Rules So Far

- So far, we specified the evaluation rules **intuitively**
 - We described how dynamic dispatch behaved in words (e.g., “just like Java”)
 - We talked about scoping, variables, arithmetic expressions (e.g., “they work as expected”)
- Why isn't this description good enough?



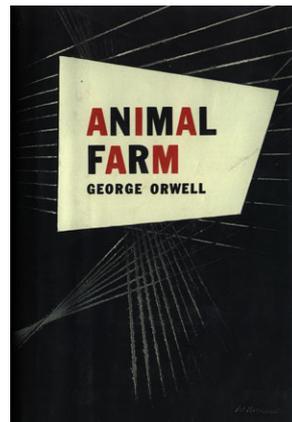
Assembly Language

Description of Semantics

- We might just tell you how to compile it
 - (but that would be helpful ...)
- But assembly-language descriptions of language implementation have too many irrelevant details
 - Which way the stack grows
 - How integers are represented on a particular machine
 - The particular instruction set of the architecture
- We need a **complete** but **not overly restrictive** specification

Programming Language Semantics

- There are many ways to specify programming language semantics
- They are all equivalent but some are more suitable to various tasks than others
- **Operational semantics**
 - Describes the evaluation of programs on an abstract machine
 - Most useful for specifying implementations
 - This is what we will use for Cool



Other Kinds of Semantics

- **Denotational semantics**

- The meaning of a program is expressed as a **mathematical object**
- Elegant but quite complicated

- **Axiomatic semantics**

- Useful for checking that programs satisfy certain correctness properties
 - e.g., that the quick sort function sorts an array
- The foundation of many **program verification systems**

Introduction to Operational Semantics

- Once again we introduce a formal notation
 - Using **logical rules of inference**, just like typing
- Recall the typing judgment

Context \vdash **e** : **T**

(in the given **context**, expression **e** has type **T**)

- We try something similar for evaluation

Context \vdash **e** : **v**

(in the given **context**, expression **e** evaluates to **value v**)

Example Operational Semantics Inference Rule

Context \vdash **e_1** : **5**

Context \vdash **e_2** : **7**

Context \vdash **$e_1 + e_2$** : **12**

- In general the result of evaluating an expression *depends on* the result of evaluating its subexpressions
- The logical rules specify everything that is needed to evaluate an expression

Aside

- The operational semantics inference rules for Cool will become quite complicated
 - i.e., many hypotheses
- This may initially look daunting
- Until you realize that the opsem rules specify exactly how to build an interpreter
- That is, every rule of inference in this lecture is pseudocode for an interpreter
 - So walking through the opsem (and thinking “I must generate code to implement this”) is a PA5 hint.

- It might be tempting to protest this excursion into Theory
- But I assert it will come in handy very soon!



What Contexts Are Needed?

- Contexts are needed to handle variables
- Consider the evaluation of $y \leftarrow x + 1$
 - We need to keep track of values of variables
 - We need to allow variables to change their values during the evaluation
- We track variables and their values with:
 - An **environment** : tells us at what address in memory is the value of a variable stored
 - A **store** : tells us what is the contents of a memory location

What Contexts Are Needed?

- Contexts are needed to handle variable

**Remind me – why do we need a separate *store* and *environment*?
Are those compiler notions?
Which is static? Which
is dynamic?**

values

- An **environment** : tells us at what address in memory is the value of a variable stored
- A **store** : tells us what is the contents of a memory location

Variable Environments

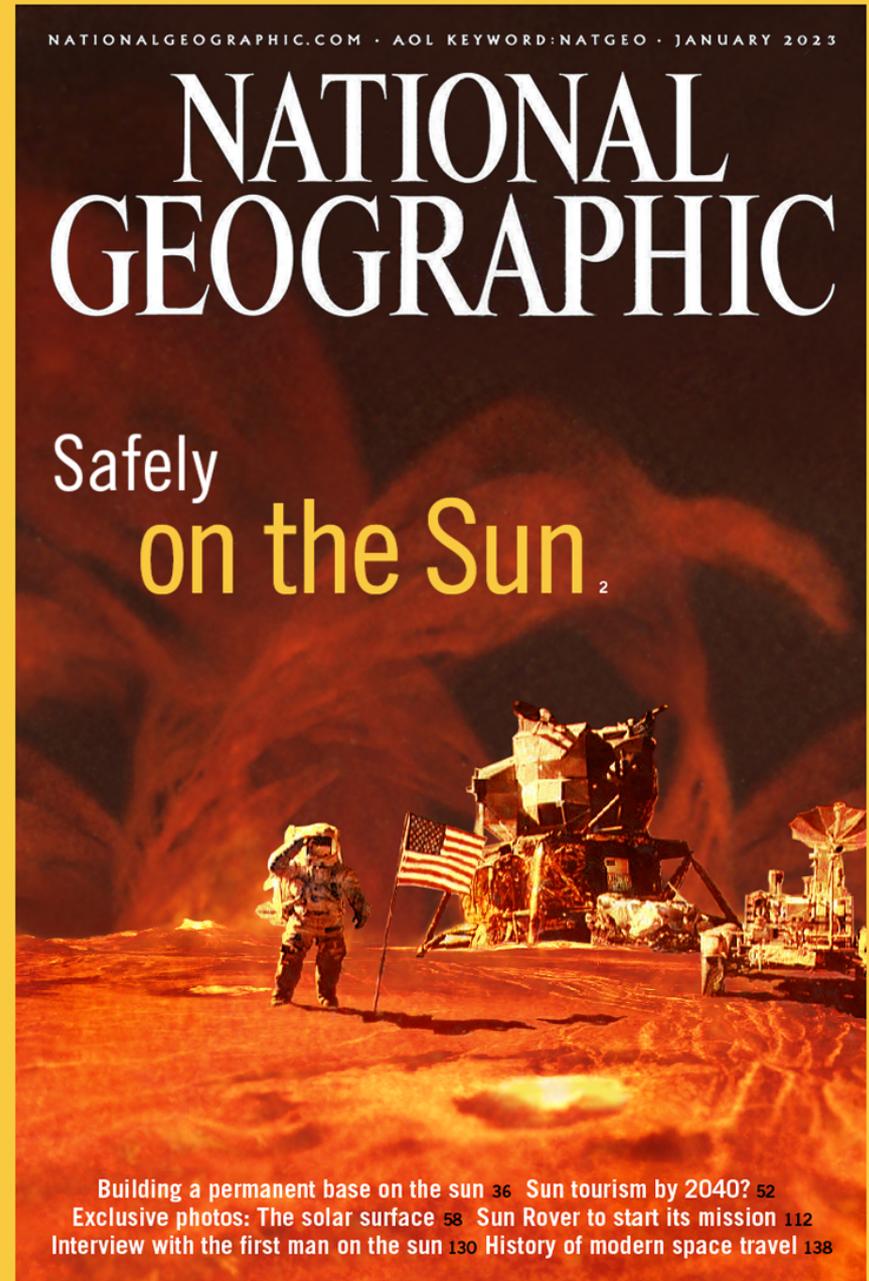
- A variable **environment** is a map from variable names to **locations**
- Tells in what memory location the value of a variable is stored
 - Locations = Memory Addresses
- Environment tracks **in-scope** variables only
- Example environment:

$$E = [a : l_1, b : l_2]$$

- To lookup a variable **a** in environment **E** we write **E(a)**

Lost?

- Environments may seem hostile and unforgiving
- But soon they'll feel just like home!
- Names → Locations



Stores

- A **store** maps memory locations to values
- Example store:

$$S = [l_1 \rightarrow 5, l_2 \rightarrow 7]$$

- To lookup the contents of a location l_1 in store S we write $S(l_1)$
- To perform an assignment of **12** to location l_1 we write $S[12/l_1]$
 - This denotes a new store S' such that
$$S'(l_1) = 12 \quad \text{and} \quad S'(l) = S(l) \text{ if } l \neq l_1$$

- Avoid mistakes in your stores!

- Locations → Values



Cool Values

- All **values** in Cool are objects
 - All objects are instances of some class (the dynamic type of the object)
- To denote a Cool object we use the notation $X(a_1 = l_1, \dots, a_n = l_n)$ where
 - X is the **dynamic type** of the object
 - a_i are the **attributes** (including those inherited)
 - l_i are the **locations** where the values of attributes are stored

Cool Values (Cont.)

- Special cases (without named attributes)
 - `Int(5)` the integer 5
 - `Bool(true)` the boolean true
 - `String(4, "Cool")` the string "Cool" of length 4
- There is a special value `void` that is a member of all types
 - No operations can be performed on it
 - Except for the test `isvoid`
 - Concrete implementations might use NULL here

Operational Rules of Cool

- The evaluation judgment is

$$so, E, S \vdash e : v, S'$$

read:

- Given so the current value of the $self$ object
- And E the current variable environment
- And S the current store
- If the evaluation of e terminates then
- The returned value is v
- And the new store is S'

Notes

- The “result” of evaluating an expression is **both** a value **and** a new store
- Changes to the store **model side-effects**
 - side-effects = assignments to variables
- The variable environment does not change
- Nor does the value of “self”
- The operational semantics allows for non-terminating evaluations
- We define one rule for each kind of expression

Operational Semantics for Base Values

$so, E, S \vdash \mathbf{true} : \mathbf{Bool}(\mathbf{true}), S$

$so, E, S \vdash \mathbf{false} : \mathbf{Bool}(\mathbf{false}), S$

i is an integer literal

$so, E, S \vdash \mathbf{i} : \mathbf{Int}(\mathbf{i}), S$

s is a string literal
 n is the length of s

$so, E, S \vdash \mathbf{s} : \mathbf{String}(n, \mathbf{s}), S$

- No side effects in these cases
(the store does not change)

Operational Semantics of Variable References

$$E(\text{id}) = l_{\text{id}}$$

$$S(l_{\text{id}}) = v$$

$$so, E, S \vdash \text{id} : v, S$$

- Note the **double lookup** of variables
 - First from name to location (compile time)
 - Then from location to value (run time)
- The store does not change
- A special case:

$$so, E, S \vdash \text{self} : so, S$$

Operational Semantics of Assignment

$$\mathbf{so, E, S \vdash e : v, S_1}$$

$$\mathbf{E(id) = l_{id}}$$

$$\mathbf{S_2 = S_1[v/l_{id}]}$$

$$\mathbf{so, E, S \vdash id \leftarrow e : v, S_2}$$

- A three step process
 - Evaluate the right hand side
 - \Rightarrow a value v and a new store S_1
 - Fetch the location of the assigned variable
 - The result is the value v and an updated store
- The environment does not change

Operational Semantics of Conditionals

$$so, E, S \vdash e_1 : \text{Bool}(\text{true}), S_1$$
$$so, E, S_1 \vdash e_2 : v, S_2$$

$$so, E, S \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : v, S_2$$

- The “**threading**” of the store enforces an evaluation sequence
 - e_1 must be evaluated first to produce S_1
 - Then e_2 can be evaluated
- The result of evaluating e_1 is a boolean object
 - The **typing rules** ensure this
 - There is another, similar, rule for **Bool(false)**

Operational Semantics of Sequences

$$so, E, S \vdash e_1 : v_1, S_1$$
$$so, E, S_1 \vdash e_2 : v_2, S_2$$
$$\dots$$
$$so, E, S_{n-1} \vdash e_n : v_n, S_n$$

$$so, E, S \vdash \{ e_1; \dots; e_n; \} : v_n, S_n$$

- Again the threading of the store expresses the intended evaluation sequence
- Only the last value is used
- But all the side-effects are collected (how?)

Q: Music (198 / 842)

- Give both of the other place names that occur in the song **Istanbul (Not Constantinople)**. It was originally performed in 1953 by **The Four Lads** and was covered by **They Might Be Giants** in 1990.

Q: Games (516 / 842)

- This 1988 entry in the King's Quest series of games was the first to feature a female protagonist. The quest involved finding the magical healing fruit and defeating an evil fairy to recover a talisman.

Q: Movies (403 / 842)

- In this 1989 comedy also starring George Carlin, the title duo collect historical figures to avoid flunking out of San Dimas High School. An indicative exchange:
"Take them to the iron maiden. / Excellent! / Execute them. / Bogus!"

Q: Books (711 / 842)

- In this 1943 Antoine de Saint-Exupery novel the title character lives on an asteroid with a rose but eventually travels to Earth.

Operational Semantics of `while` (1)

$$\frac{\text{so, E, S} \vdash e_1 : \text{Bool}(\text{false}), S_1}{\text{so, E, S} \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \text{void}, S_1}$$

- If e_1 evaluates to `Bool(false)` then the loop terminates immediately
 - With the side-effects from the evaluation of e_1
 - And with (arbitrary) result value `void`
- The typing rules ensure that e_1 evaluates to a boolean object

Operational Semantics of `while` (2)

$$\text{so, } \mathbf{E, S} \vdash \mathbf{e_1 : Bool(true), S_1}$$
$$\text{so, } \mathbf{E, S_1} \vdash \mathbf{e_2 : v, S_2}$$
$$\text{so, } \mathbf{E, S_2} \vdash \mathbf{while\ e_1\ loop\ e_2\ pool : void, S_3}$$

$$\text{so, } \mathbf{E, S} \vdash \mathbf{while\ e_1\ loop\ e_2\ pool : void, S_3}$$

- Note the sequencing ($S \rightarrow S_1 \rightarrow S_2 \rightarrow S_3$)
- Note how looping is expressed
 - Evaluation of “`while ..`” is expressed in terms of the evaluation of `itself` in another state
- The result of evaluating `e2` is discarded
 - Only the side-effect is preserved

Operational Semantics of **let** Expressions (1)

$$so, E, S \vdash e_1 : v_1, S_1$$
$$so, ?, ? \vdash e_2 : v, S_2$$

$$so, E, S \vdash \mathbf{let\ id : T \leftarrow e_1\ in\ e_2 : v_2, S_2}$$

- What is the context in which e_2 must be evaluated?
 - Environment like E but with a new binding of id to a fresh location l_{new}
 - Store like S_1 but with l_{new} mapped to v_1

Operational Semantics of **let** Expressions (II)

- We write $l_{\text{new}} = \text{newloc}(S)$ to say that l_{new} is a location that is not already used in S
 - Think of **newloc** as the dynamic memory allocation function (or reserving stack space)
- The operational rule for **let**:

$$\frac{\begin{array}{c} \mathbf{so, E, S \vdash e_1 : v_1, S_1} \\ l_{\text{new}} = \mathbf{newloc}(S_1) \\ \mathbf{so, E[l_{\text{new}}/id], S_1[v_1/l_{\text{new}}] \vdash e_2 : v_2, S_2} \end{array}}{\mathbf{so, E, S \vdash \text{let id : T} \leftarrow e_1 \text{ in } e_2 : v_2, S_2}}$$

Balancing Act

- Now we're going to do some **very difficult** rules
 - new, dispatch
- This may initially seem tricky
 - How could that possibly work?
 - What's going on here?
- With time, these rules can actually be elegant!

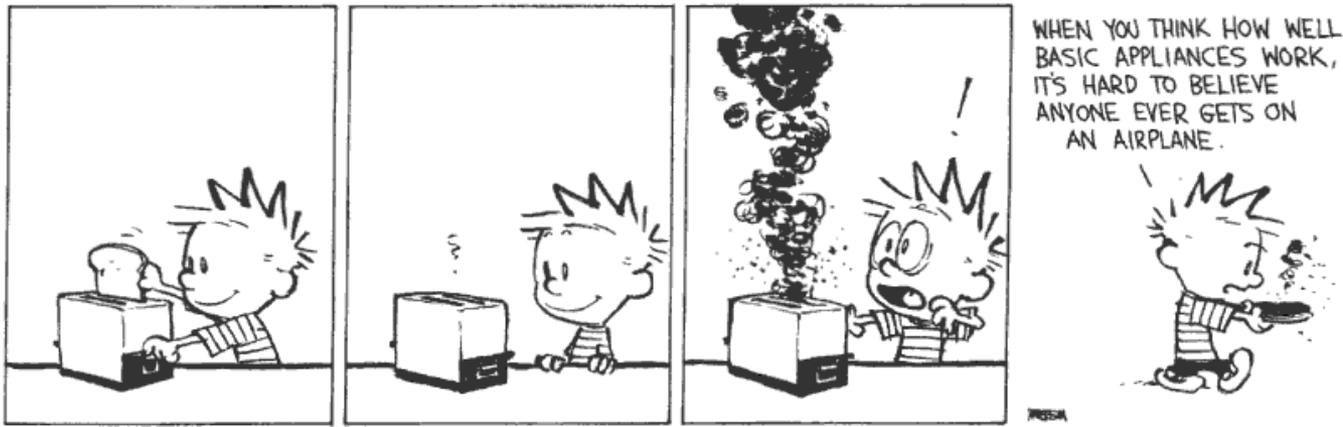


Operational Semantics of `new`

- Consider the expression `new T`
- Informal semantics
 - Allocate new locations to hold the values for all attributes of an object of class `T`
 - Essentially, allocate space for a new object
 - Initialize those locations with the default values of attributes
 - Evaluate the initializers and set the resulting attribute values
 - Return the newly allocated object

Default Values

- For each class A there is a default value denoted by D_A
 - $D_{\text{int}} = \text{Int}(0)$
 - $D_{\text{bool}} = \text{Bool}(\text{false})$
 - $D_{\text{string}} = \text{String}(0, \text{""})$
 - $D_A = \text{void}$ (for all others classes A)



More Notation

- For a class A we write

$$\text{class}(A) = (a_1 : T_1 \leftarrow e_1, \dots, a_n : T_n \leftarrow e_n)$$

where

- a_i are the attributes (including inherited ones)
 - T_i are their declared types
 - e_i are the initializers
- This is the **class map** from PA4!

Operational Semantics of **new**

- Observation: **new SELF_TYPE** allocates an object with the same dynamic type as **self**

$T_0 =$ if $T == \text{SELF_TYPE}$ and $so = X(\dots)$ then X else T

$\text{class}(T_0) = (a_1 : T_1 \leftarrow e_1, \dots, a_n : T_n \leftarrow e_n)$

$l_i = \text{newloc}(S)$ for $i = 1, \dots, n$

$v = T_0(a_1 = l_1, \dots, a_n = l_n)$

$S_1 = S[D_{T_1}/l_1, \dots, D_{T_n}/l_n]$

$E' = [a_1 : l_1, \dots, a_n : l_n]$

$v, E', S_1 \vdash \{ a_1 \leftarrow e_1; \dots; a_n \leftarrow e_n; \} : v_n, S_2$

*Initialize
new object*

$so, E, S \vdash \text{new } T : v, S_2$

Operational Semantics of **new**

- Observation: **new SELF_TYPE** allocates an object with the same dynamic type as **self**

$T_0 =$ if $T == \text{SELF_TYPE}$ and $so = X(\dots)$ then X else T

$\text{class}(T_0) = (a_1 : T_1 \leftarrow e_1, \dots, a_n : T_n \leftarrow e_n)$

$l_i = \text{newloc}(S)$ for $i = 1, \dots, n$

$v = T_0(a_1 = l_1, \dots, a_n = l_n)$

$S_1 = S[D_{T_1}/l_1, \dots, D_{T_n}/l_n]$

$E' = [a_1 : l_1, \dots, a_n : l_n]$

$v, E', S_1 \vdash \{ a_1 \leftarrow e_1; \dots; a_n \leftarrow e_n; \} : v_n, S_2$

*Initialize
new object*

$so, E, S \vdash \text{new } T : v, S_2$

Operational Semantics of `new`

- The first three lines `allocate` the object
- The rest of the lines `initialize` it
 - By evaluating a sequence of assignments
- State in which the initializers are evaluated:
 - Self is the current object
 - Only the attributes are in scope (same as in typing)
 - Starting value of attributes are the default ones
- Side-effects of initialization are kept (in \mathbf{s}_2)

Operational Semantics of Method Dispatch

- Consider the expression $e_0.f(e_1, \dots, e_n)$
- Informal semantics:
 - Evaluate the arguments in order e_1, \dots, e_n
 - Evaluate e_0 to the target object
 - Let X be the **dynamic** type of the target object
 - Fetch from X the definition of f (with n args)
 - Create n new locations and an environment that maps f 's formal arguments to those locations
 - Initialize the locations with the actual arguments
 - Set **self** to the target object and evaluate f 's body

More Notation

- For a class A and a method f of A (possibly inherited) we write:

$$\text{imp}(A, f) = (x_1, \dots, x_n, e_{\text{body}})$$

where

- x_i are the names of the **formal arguments**
 - e_{body} is the **body** of the method
- This is the **imp map** from PA4!

Dispatch OpSem

$so, E, S \vdash e_1 : v_1, S_1$

$so, E, S_1 \vdash e_2 : v_2, S_2$

...

$so, E, S_{n-1} \vdash e_n : v_n, S_n$

$so, E, S_n \vdash e_0 : v_0, S_{n+1}$

$v_0 = X(a_1 = l_1, \dots, a_m = l_m)$

$imp(X, f) = (x_1, \dots, x_n, e_{body})$

$l_{xi} = newloc(S_{n+1})$ for $i = 1, \dots, n$

$E' = [x_1 : l_{x1}, \dots, x_n : l_{xn}, a_1 : l_1, \dots, a_m : l_m]$

$S_{n+2} = S_{n+1}[v_1/l_{x1}, \dots, v_n/l_{xn}]$

$v_0, E', S_{n+2} \vdash e_{body} : v, S_{n+3}$

Evaluate arguments

Evaluate receiver object

Find type and attributes

Find formals and body

New environment

New store

Evaluate body

$so, E, S \vdash e_0.f(e_1, \dots, e_n) : v, S_{n+3}$

Operational Semantics of Dispatch

- The body of the method is invoked with
 - **E** mapping formal arguments and self's attributes
 - **S** like the caller's except with actual arguments bound to the locations allocated for formals
- The notion of the activation frame is implicit
 - New locations are allocated for actual arguments
- The semantics of static dispatch is similar except the implementation of **f** is taken from the specified class

Runtime Errors

Operational rules do not cover all cases

Consider for example the rule for dispatch:

...

$\mathbf{so, E, S_n \vdash e_0 : v_0, S_{n+1}}$

$\mathbf{v_0 = X(a_1 = l_1, \dots, a_m = l_m)}$

$\mathbf{imp(X, f) = (x_1, \dots, x_n, e_{body})}$

...

$\mathbf{so, E, S \vdash e_0.f(e_1, \dots, e_n) : v, S_{n+3}}$

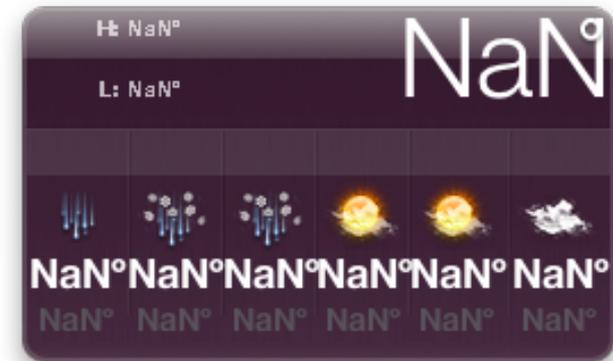
What happens if $\mathbf{imp(X, f)}$ is not defined?

Cannot happen in a well-typed program

(because of the Type Safety Theorem)

Runtime Errors

- There are some runtime errors that the type checker does not try to prevent
 - A dispatch on void
 - **Division by zero**
 - Substring out of range
 - Heap overflow
- In such case the execution must abort gracefully
 - With an error message and not with a segfault



Conclusions

- Operational rules are very **precise**
 - Nothing is left unspecified
- Operational rules contain a lot of details
 - Read them **carefully**
- Most languages do not have a well specified operational semantics
- When portability is important an operational semantics becomes essential
 - But not always using the exact notation we used for Cool

Homework

- PA4 due Wed Oct 28
- WA5 due Thu Oct 29
- Next:
 - Chapters 9.4 - 9.7 in course book
 - Optional Wikipedia articles