Operational Semantics

One-Slide Summary

- **Operational semantics** is 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

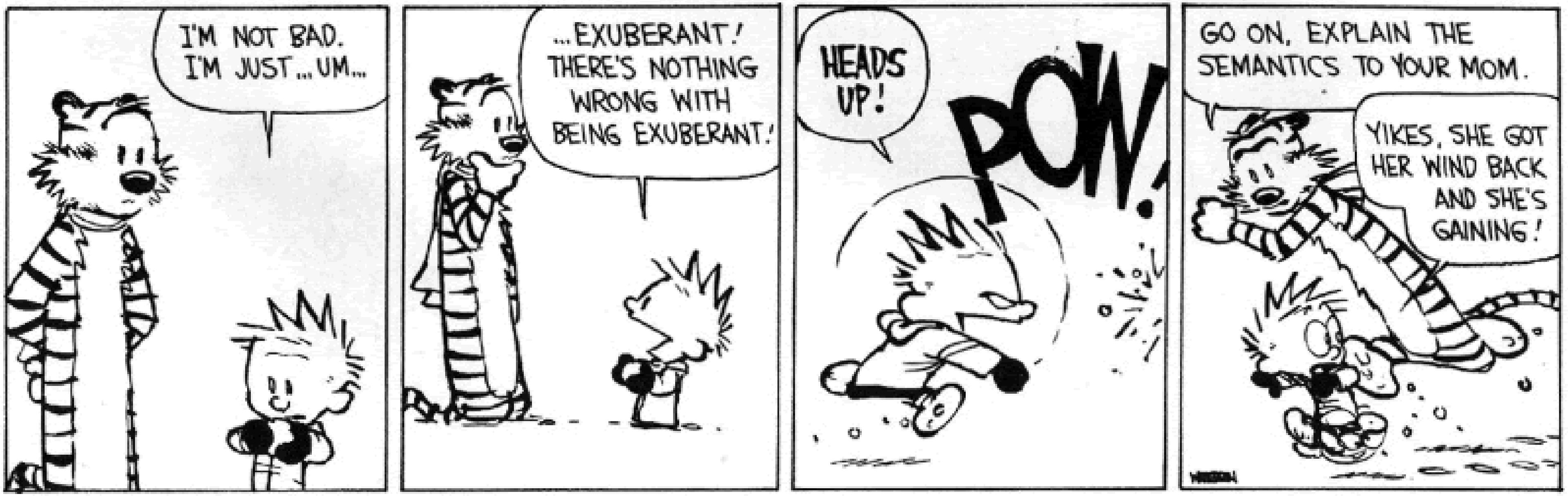
- The definition of a programming language:
  - The tokens ⇒ lexical analysis
  - The grammar ⇒ syntactic analysis
  - The typing rules ⇒ semantic analysis
  - The evaluation rules ⇒ interpretation

    (also: staged 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?

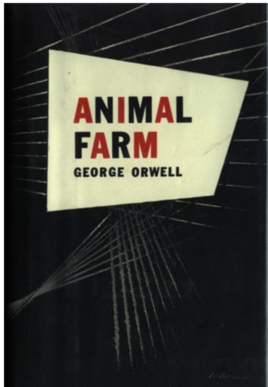

I'M NOT BAD. I'M JUST...UM...
...EXUBERANT! THERE'S NOTHING WRONG WITH BEING EXUBERANT!
HEADS UP!
POW!
GO ON, EXPLAIN THE SEMANTICS TO YOUR MOM.
YIKES, SHE GOT HER WIND BACK AND SHE'S GAINING!

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, cf. typing rules
- Recall the typing judgment

$$\text{Context} \vdash e : T$$

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

- We try something similar for evaluation

$$\text{Context} \vdash e : v$$

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

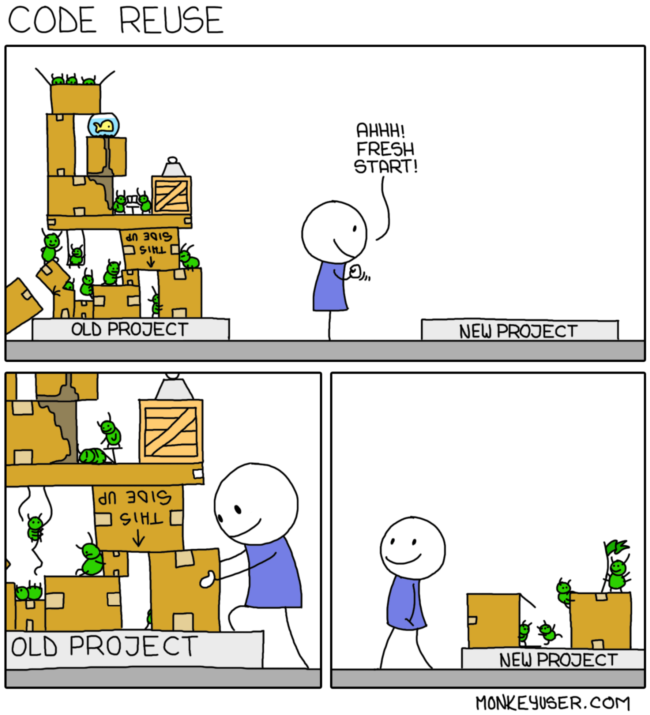
Example Operational Semantics Inference Rule

$$\frac{\textbf{Context} \vdash e_1 : 5 \qquad \textbf{Context} \vdash e_2 : 7}{\textbf{Context} \vdash e_1 + e_2 : 12}$$

- 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

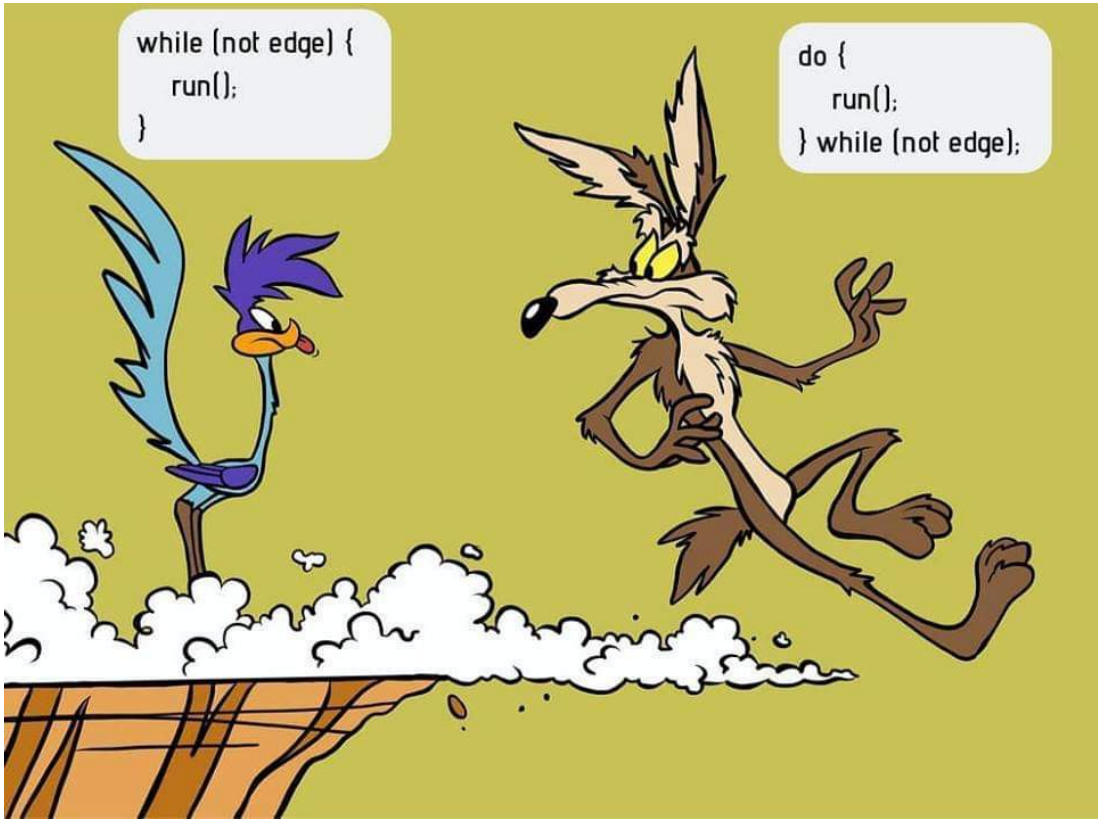
Aside

- The operational semantics inference rules for Cool will become 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



Theory Details Matter

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



```
while (not edge) {
    run();
}
```

```
do {
    run();
} while (not edge);
```

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 variab s

Remind me – why do we need a separate *store* and *environment*?

Which is static?
Which is dynamic?

es

n:

  - 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

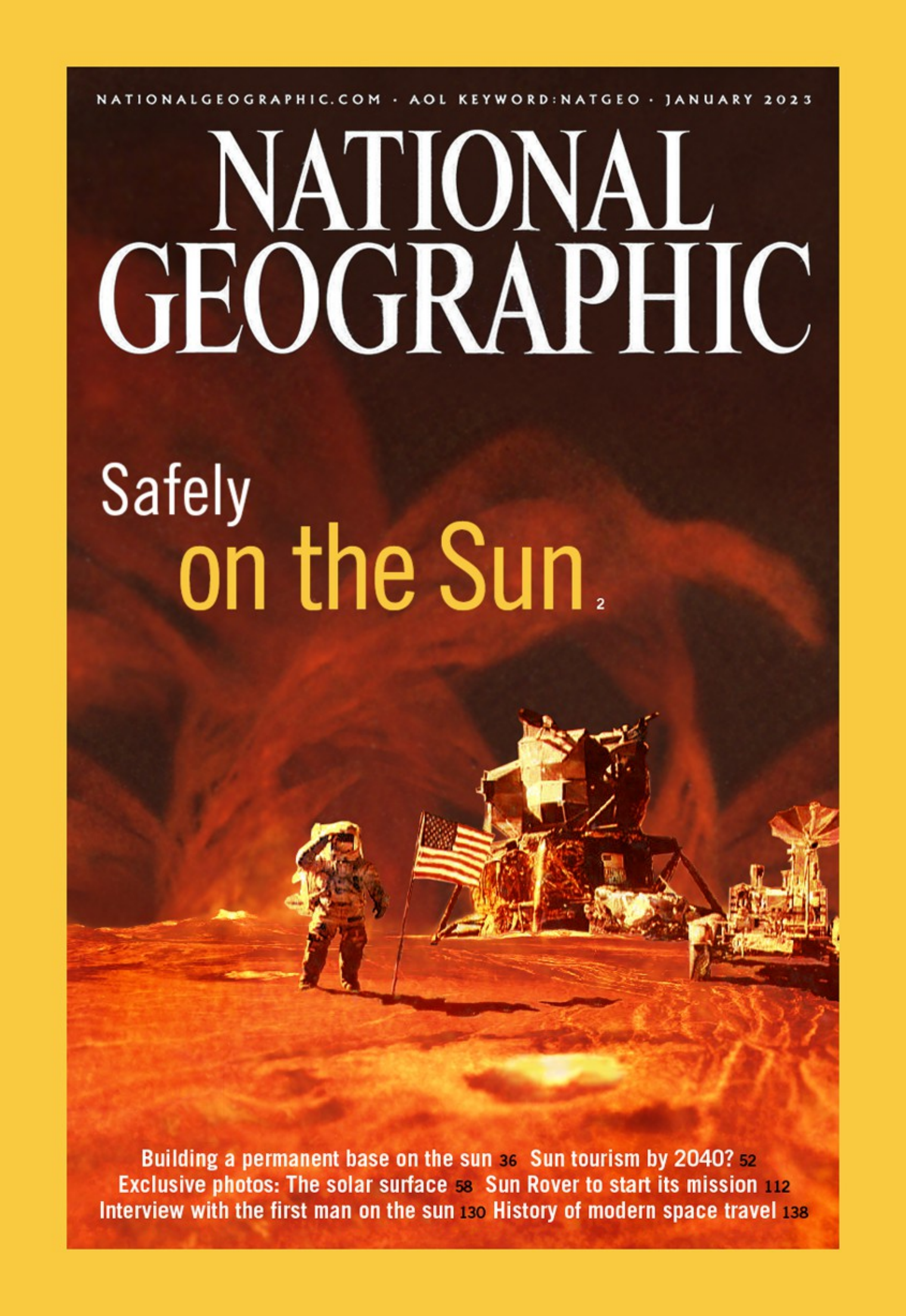
- A variable **environment** is a map from variable names to **locations**
- Tells in what memory location the value of a variable is stored
  - Locations = Memory Addresses
- Environment tracks in-scope variables only
- Example environment:

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

- 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


NATIONALGEOGRAPHIC.COM · AOL KEYWORD:NATGEO · JANUARY 2023
NATIONAL GEOGRAPHIC
Safely on the Sun 2
Building a permanent base on the sun 36 Sun tourism by 2040? 52
Exclusive photos: The solar surface 58 Sun Rover to start its mission 112
Interview with the first man on the sun 130 History of modern space travel 138

Stores

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

$$S = [l_1 \to 5, \;\; l_2 \to 7]$$

- To lookup the contents of a location l_1 in store S we write $S(l_1)$
- To perform an assignment of 23 to location l_1 we write $S[23/l_1]$
  - This denotes a new store S' such that

$$S'(l_1) = 23 \quad \text{and} \quad S'(l) = S(l) \text{ if } l \neq l_1$$

- 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 $X(a_1 = l_1, \ldots, a_n = l_n)$ where
  - X is the dynamic type of the object (type tag)
  - 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

  - Last lecture: Int(5) and String(4,“Cool”) in Ocaml

- 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 also a new store
- Changes to the store model side-effects
  - side-effects = assignments to mutable 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

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

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

$$\frac{\text{i is an integer literal}}{\text{so, E, S} \vdash \text{i} : \text{Int(i), S}}$$

$$\frac{\begin{array}{c}\text{s is a string literal}\\ \text{n is the length of s}\end{array}}{\text{so, E, S} \vdash \text{s} : \text{String(n,s), S}}$$

- No side effects in these cases

  (the store does not change)

Operational Semantics of Variable References

$$\frac{\begin{array}{c} E(id) = l_{id} \\ S(l_{id}) = v \end{array}}{so, E, S \vdash id : v, S}$$

- 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:

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

Operational Semantics of Assignment

$$\frac{\begin{array}{c}\mathbf{so, E, S \vdash e : v, S_1} \\ \mathbf{E(id) = l_{id}} \\ \mathbf{S_2 = S_1[v/l_{id}]}\end{array}}{\mathbf{so, E, S \vdash id \leftarrow e : v, S_2}}$$

- A three step process
  - Evaluate the right hand side
    - ⇒ 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

$$\frac{\begin{array}{c}so, E, S \vdash e_1 : Bool(true), S_1 \\ so, E, S_1 \vdash e_2 : v, S_2\end{array}}{so, E, S \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : v, S_2}$$

- 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

$$\frac{\begin{array}{c}\textbf{so, E, S} \vdash \textbf{e}_1 : \textbf{v}_1\textbf{, S}_1 \\ \textbf{so, E, S}_1 \vdash \textbf{e}_2 : \textbf{v}_2\textbf{, S}_2 \\ \ldots \\ \textbf{so, E, S}_{n-1} \vdash \textbf{e}_n : \textbf{v}_n\textbf{ , S}_n\end{array}}{\textbf{so, E, S} \vdash \{\ \textbf{e}_1\textbf{; ...; e}_n\textbf{; }\} : \textbf{v}_n\textbf{, S}_n}$$

- 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: 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.

Fairy Tales Trivia

(student "memorial")

- Identify the authors of **Kinder- und Hausmärchen**, a German collection of fairy tales from 1812 that includes:
  - Rapunzel
  - Hansel and Gretel
  - Cinderella
  - Rumpelstiltskin
  - Golden Goose
  - Little Snow White

Kinder-

und

Haus-Märchen

Erſter Theil.

Geography Trivia

(student “memorial”)

- *This* is traditionally considered the longest river in the world (although some suggest that the Amazon is longer). It flows north and is about 6,650 km (4,130 mi) long. A “stone” critical to deciphering scripts was found in its port town of Rashid.

Real-World Languages

- This North Germanic language was spoken by Scandinavians and their overseas settlements until about 1300. It featured masculine, feminine and neuter genders, with adjectives and pronouns mirroring the gender of the antecedent noun. English adopts words such as slaughter (slátr), thrift (þrift) and anger (angr) from this.

Operational Semantics of while (1)

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

- 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)

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

- 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 e_2 is discarded
  - Only the side-effect is preserved

Operational Semantics of let Expressions (1)

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : v_1, S_1 \\ so, ?, ? \vdash e_2 : v, S_2 \end{array}}{so, E, S \vdash \mathbf{let}\ id : T \leftarrow e_1\ \mathbf{in}\ e_2 : v_2, S_2}$$

- 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_{new} = 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:

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

Balancing Act

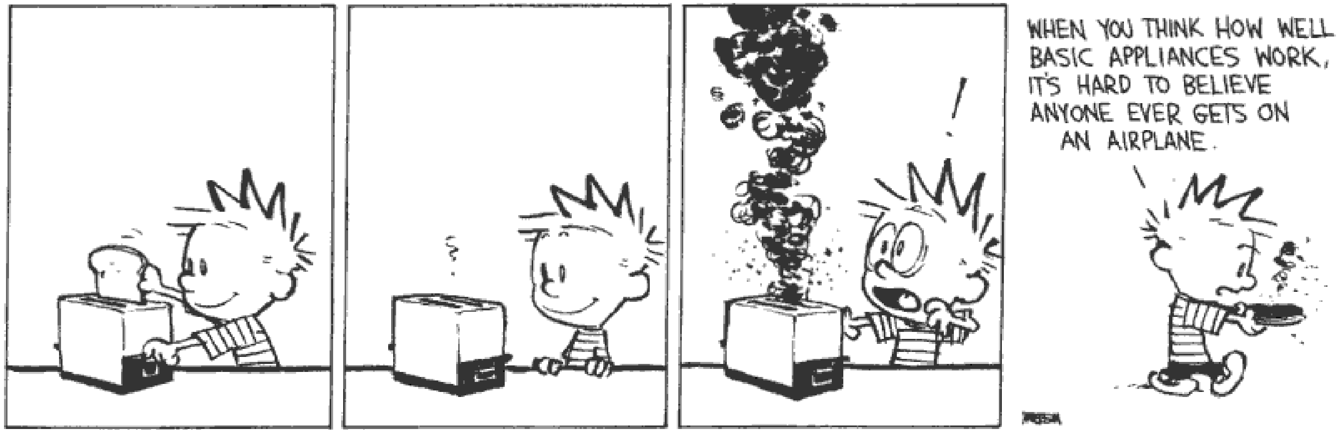
- Now we're going to do some very difficult rules
  - new, dispatch
- This may initially seem tricky
  - How could that possibly work?
  - What's going on here?
- With time, these rules can actually be elegant!

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_{int} = Int(0)
  - D_{bool} = Bool(false)
  - D_{string} = String(0, “”)
  - D_A = void (for all others classes A)


!
WHEN YOU THINK HOW WELL BASIC APPLIANCES WORK, IT'S HARD TO BELIEVE ANYONE EVER GETS ON AN AIRPLANE.

More Notation

- For a class A we write

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

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

$$\begin{array}{l}
T_0 = \text{if } T == \text{SELF\_TYPE and } so = X(\ldots) \text{ then } X \text{ else } T \\
class(T_0) = (a_1 : T_1 \leftarrow e_1, \ldots, a_n : T_n \leftarrow e_n) \\
l_i = newloc(S) \text{ for } i = 1, \ldots, n \\
\left.\begin{array}{l}
v = T_0(a_1 = l_1, \ldots, a_n = l_n) \\
S_1 = S[D_{T1}/l_1, \ldots, D_{Tn}/l_n] \\
E' = [a_1 : l_1, \ldots, a_n : l_n] \\
v, E', S_1 \vdash \{\, a_1 \leftarrow e_1; \ldots; a_n \leftarrow e_n; \,\} : v_n, S_2
\end{array}\right\} \textit{Initialize new object} \\
\hline
so, E, S \vdash \text{new } T : v, S_2
\end{array}$$

Operational Semantics of new

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

$$\dfrac{\begin{array}{l}
T_0 = \text{if } T == \text{SELF\_TYPE and } so = X(\ldots) \text{ then } X \text{ else } T\\
class(T_0) = (a_1 : T_1 \leftarrow e_1, \ldots, a_n : T_n \leftarrow e_n)\\
l_i = newloc(S) \text{ for } i = 1,\ldots,n\\
\left.\begin{array}{l}
v = T_0(a_1 = l_1,\ldots,a_n = l_n)\\
S_1 = S[D_{T1}/l_1,\ldots,D_{Tn}/l_n]\\
E' = [a_1 : l_1, \ldots, a_n : l_n]\\
v, E', S_1 \vdash \{ a_1 \leftarrow e_1; \ldots; a_n \leftarrow e_n; \} : v_n, S_2
\end{array}\right\} \textit{Initialize new object}
\end{array}}{so, E, S \vdash \mathbf{new}\ T : v, S_2}$$

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 S_2)

Operational Semantics of Method Dispatch

- Consider the expression $e_0.f(e_1,…,e_n)$
- Informal semantics:
  - Evaluate the arguments in order $e_1,…,e_n$
  - 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:

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

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

$$\dfrac{\begin{array}{ll}
\left.\begin{array}{l}
so, E, S \vdash e_1 : v_1, S_1 \\
so, E, S_1 \vdash e_2 : v_2, S_2 \\
\ldots \\
so, E, S_{n-1} \vdash e_n : v_n, S_n
\end{array}\right\} & \textit{Evaluate arguments} \\
\left.so, E, S_n \vdash e_0 : v_0, S_{n+1}\right\} & \textit{Evaluate receiver object} \\
\left.v_0 = X(a_1 = l_1, \ldots, a_m = l_m)\right\} & \textit{Find type and attributes} \\
\left.imp(X, f) = (x_1, \ldots, x_n, e_{body})\right\} & \textit{Find formals and body} \\
l_{xi} = newloc(S_{n+1}) \text{ for } i = 1, \ldots, n & \\
\left.E' = [x_1 : l_{x1}, \ldots, x_n : l_{xn}, a_1 : l_1, \ldots, a_m : l_m]\right\} & \textit{New environment} \\
\left.S_{n+2} = S_{n+1}[v_1/l_{x1}, \ldots, v_n/l_{xn}]\right\} & \textit{New store} \\
\left.v_0, E', S_{n+2} \vdash e_{body} : v, S_{n+3}\right\} & \textit{Evaluate body}
\end{array}}{so, E, S \vdash e_0.f(e_1,\ldots,e_n) : v, S_{n+3}}$$

Dispatch OpSem (scoping detail)

$$
\frac{
\begin{array}{l}
so, E, S \vdash e_1 : v_1, S_1 \\
so, E, S_1 \vdash e_2 : v_2, S_2 \\
\ldots \\
so, E, S_{n-1} \vdash e_n : v_n, S_n \\
so, E, S_n \vdash e_0 : v_0, S_{n+1} \\
v_0 = X(a_1 = l_1, \ldots, a_m = l_m) \\
imp(X, f) = (x_1, \ldots, x_n, e_{body}) \\
l_{xi} = newloc(S_{n+1}) \text{ for } i = 1, \ldots, n \\
E' = [\overbrace{x_1 : l_{x1}, \ldots, x_n : l_{xn}}^{\textit{Formal Parameters}}, \overbrace{a_1 : l_1, \ldots, a_m : l_m}^{\textit{Fields (shadowed by params)}}] \\
S_{n+2} = S_{n+1}[v_1/l_{x1}, \ldots, v_n/l_{xn}] \\
v_0, E', S_{n+2} \vdash e_{body} : v, S_{n+3}
\end{array}
}{
so, E, S \vdash e_0.f(e_1, \ldots, e_n) : v, S_{n+3}
}
$$

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:

$$\frac{\begin{array}{l} \ldots \\ so, E, S_n \vdash e_0 \ : v_0, S_{n+1} \\ v_0 = X(a_1 = l_1, \ldots, a_m = l_m) \\ imp(X, f) = (x_1, \ldots, x_n, e_{body}) \\ \ldots \end{array}}{so, E, S \vdash e_0.f(e_1, \ldots, e_n) \ : v, S_{n+3}}$$

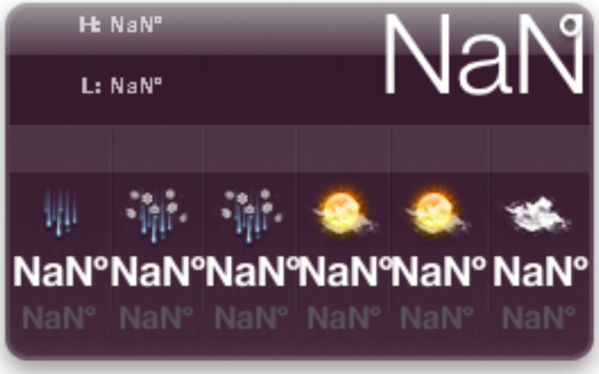
What happens if imp(X, f) is not defined?

Cannot happen in a well-typed program

(because of the Type Safety Theorem, yay)

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 (pushed back to Today)
- PA5t Tuesday

- RS5 Recommended April 4th