Applications of operational semantics

Would a consideration of operational semantics have any practical effect on one’s coding?

Yes, but probably at one level of remove: you use operational semantics to be sure you understand what is supposed to happen when you run the code, and then you use your understanding to write the code you want. The primary way I expect you to use operational semantics is to analyze a language you have never seen before. Operational semantics can also help guide you to good questions:

- In Python, what happens if I assign to an unbound variable inside a block?
- In Lua, what happens if the number of actual parameters I pass to a function doesn’t match the number of formal parameters the function is expecting?

As you’ll use it, operational semantics is more of a vocabulary than anything else. If you’re going to analyze a language you’ve never seen before, you need a vocabulary with which to think about your analysis.

Do I really need to know this stuff if I’m not designing new programming languages?

Only if you want to understand a language that somebody else has designed. (It is quite possible to be productive in a language without understanding what you are saying. But we prefer that our Tufts Jumbos actually know what they are saying.)

Can operational semantics be applied to every valid [programming] language?

Yes. If you can run it, or if you can write an interpreter for it, then operational semantics can be applied to it.

Can operational semantics be used to describe any type of language? or just functional/deterministic ones?

Any type. On page 892, for example, you’ll find an operational semantics for the object-oriented language Smalltalk. Operational semantics is especially good at expressing non-deterministic languages, although such languages are beyond the scope of 105.

Does it apply to all languages? How is it useful in real life programming experience?

It’s useful for figuring out tricky cases.

How practical are operational semantics for very complex languages?

Operational semantics for a whole language is usually expensive, and the engineering benefits are rarely worth the cost. What you’ll see in practical use is typically an operational semantics for some fragment of a language or some tricky part of a language design whose meaning is not clear. For a good example of the practical application of operational semantics to the very complex, real language Haskell, check out Simon Peyton Jones’s tutorial “Tackling the Awkward Squad”. You’ll be well equipped to read this once we finish with a language called “lambda calculus.”

To what extent are the operational semantics we’ve learned around Impcore extendable to other languages and how do those linguistic differences impact the Opsem we employ? (What are some examples?)

Because they don’t support pointers, the Impcore semantics are fairly limited; they are just a starting point. When we shift over to the location semantics mentioned in class today, things get a lot more extendable—you can start to express ideas like “take the address of a local variable,” for example. Location semantics is also good for the likes of Python, JavaScript, C, C++, and so on. As for how linguistic differences affect semantics, the big changes are less around the language as a whole and more around what features you want to account for. Here are a couple of examples:

- If you want to account for throwing exceptions or for checked run-time errors, you need something more than a value in the final state.
- If you want to account for I/O (printing and suchlike) done by computations that haven’t yet terminated, you probably want something called a “small-step” semantics, where the evaluation judgment tracks every step of a computation, not just the final result.
- If you want to account for the behavior of computations that might not terminate (like a web server or an operating system), you definitely use small-step semantics.

There is a small-step semantics for Scheme in Chapter 3 of the book, which I can get for you if you like, though we won’t study it this term. We will study small-step semantics in the context of “lambda calculus.”

Applications of metatheory

What are interesting metatheoretic proofs that have been done for real languages and in what context are they useful? Are there ways of automating them?

Off the top of my head:

- Microsoft constructed a proof that a Windows device driver written in Sing# can’t take down the kernel—the worst that can happen is you lose the device.
- There’s been a lot of work on “information-flow” security, all of the flavor “information from a high-classified source never affects low-classified output.” These theorems are of great interest to certain governmental agencies.
- The classic PL metatheorem is that a type system actually says something about what happens at run time.
- There’s been some very interesting work recently at Boeing using language technology to secure the flight software for a helicopter. One impressive result: a red team was given full source code for the software, plus root access to the system on which the software was running, and yet they still were not able to compromise the flight software. This feat was accomplished using proof techniques—I don’t know the details, but Kathleen Fisher would know.

Doing any of these proofs at full scale for a real language still requires heroic effort. Automating proofs is done using heuristics called “tactics,” but this is still a topic of active research. If you want to study this area, we have a world expert (Adam Chlipala) across town at MIT.

Philosophy: What and why

Q: Is the operational semantics the derivation? A: I think derivation is a way to prove the correctness of operational semantics.

If we say “the semantics” (as for Impcore), we mean the judgment forms and proof rules for a semantic system for a particular language or language feature. If we say “operational semantics,” we mean a whole body of techniques for explaining what happens when we run the code, usually in contrast to “denotational semantics” and “axiomatic semantics.”

A derivation proves that evaluation terminates, plus what evaluation produces.

It is not meaningful to talk about “correctness of semantics.” A semantics is just a bunch of rules, and it is no more correct or incorrect than the rules for Fizzbin, or the

http://memory-alpha.wikia.com/wiki/Fizzbin
rules for Fluxx. (Fluxx is a very algorithmic card game.) With operational semantics, we are more concerned with questions like “does it say what we intended?” (does it accurately describe the feature we hoped for), “is it useful” (can we prove anything of interest), or “is it faithful to the implementation?”

In class today, when you asked about semantic flaws, someone answered something, and you said “well that’s more about the rules.” Is it then the case that Rules = Operational Semantics and semantics in general is something different? What distinguishes the two?

Rules are part of a semantics. In class I was focusing on judgment forms, which limit what rules we can write. Distinguish them this way: forms of judgment limit what we can say, where rules determine of all the things we can say, which ones are true.

“Semantics in general” would be any (formal) way of saying what code is supposed to mean.

Why??

Theory: to know what code is supposed to do when you run it.

Metatheory: to know the provable strengths and limitations of a language.

Doing metatheory

How do we use a derivation tree as a way to prove two statements expressions are equivalent?

You prove a metatheorem: When evaluated in equivalent initial states, the expressions produce equivalent results (and possibly equivalent final states). This idea is related to a common theorem called “observational equivalence.”

I don’t understand how there isn’t a distinction between base cases and inductive cases with sub-derivations when proving a metatheorem case-by-case. Doesn’t the induction have to stop somewhere (i.e. at a base case)?

My bad. There is totally such a distinction, and you are right that the induction has to stop somewhere. (A mathematician would say “well-founded.”) What I’m claiming is that when it comes time to actually write a metatheoretic proof, the distinction doesn’t help.

How do we know our cases in a metatheoretic proof?

In a metatheoretic proof, you need one case for each rule of the operational semantics.

Can you explain what separate cases are necessary in a proof?

One for each rule.

For metatheoretic proofs, is it enough to just say “by inductive hypothesis” as you do in the slides?

I want to know to which subderivation the inductive hypothesis is applied and what conclusion is justified by the inductive hypothesis. I hope I am doing this on the slides!!

In a metatheoretic proof (like question 20) how do we make the jump from, this is the rule, this is the induction hypothesis, this rule is true for this hypothesis? Especially when doing rules w/no conditionals, e.g. literals are deterministic.

I’m not sure I can help here, but try this:

1. **This is the rule.** You know the rule by assumption. Every valid derivation has to end in some rule, so you assume it must be on the list, and you do a case in the proof for each one on the list.

2. **This is the induction hypothesis.** The induction hypothesis never changes. A useful induction hypothesis tells you something about the derivations of the evaluation judgments that appear above the line (in the current rule). Thus, the hypothesis holds not for “this rule” but for a subderivation of a judgment above the line.

   Sometimes you can’t prove a theorem unless you find an induction hypothesis that is stronger than the theorem itself.

3. **Rules with no conditionals.** Here you’ve misunderstood something, but I can’t tell exactly what. Once you’ve picked a rule (not a syntactic form), what’s in the rule is determined. That’s why there are multiple rules for some forms (If, Var, and so on).

   I think what you’re looking for might be “what if a rule has no judgments above the line”? In that case, the induction hypothesis is useless, and you have to look for some other way to get the proof through.

How do you distinguish between a premise that you can apply the inductive hypothesis to, and one you can’t?

If the case you’re looking at has an evaluation judgment above the line, you can apply the inductive hypothesis to the derivation of that judgment (and to every evaluation judgment that appears above the line).

I think metatheory is still a little suspicious about the general form and how to properly execute one proof?

Have a close look at page 61 and the example proof that follows.

Making metatheory proofs readable but complete?

Again, page 61.

What does the base case “$D_r$” actually look like in the inductive step of a metatheoretical proof?

---

It is a name for a valid subderivation. In a proof, it is not necessary to look at the subderivation. In another setting, it would look like a complete, valid derivation tree for a subexpression.

**What new thing am I learning by proving the homework metatheory question?**

The most important thing you're starting to learn is, “what kind of thing can we know about every computation, and how can we know it?” You may also learn something about how to identify unintentional nondeterminism (a.k.a. “ambiguity”) in a semantics.

Is it enough when you just have \( \langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle \) to say that \( e \) is deterministic because it will always evaluate to \( v \) under these environments?

Not if you’re trying to prove anything. Saying “\( e \) will always evaluate to \( v \)” is a claim without proof.

*I don’t exactly have a specific question and this is more of an office hours thing, but I am still not sure I understand what you want out of our proofs. There is a disconnect between the derivation trees and induction for me.*

If page 61 and the following section don’t help, then yes, this is probably an office-hours thing.

**Theory vs metatheory**

*When to use theory, when to use metatheory? (With an example, if possible. Thank you!)*

Theory to show that a particular computation does a particular thing. Metatheory to show that every terminating computation has some property:

- Theory: if \( \rho(x) = 7 \), then \( (if \ x \ x \ 3) \) evaluates to 7.
- Theory: the value of \( (if \ x \ x \ 3) \) depends only on \( x \) and not on the values of other variables.
- Metatheory (false conjecture): the result of evaluating any expression \( e \) depends only on the values of the variables that are mentioned in \( e \)'s source code.

**From semantics to implementation**

*I don’t exactly have a specific question and this is more of an office hours thing, but I am still not sure I understand what you want out of our proofs. There is a disconnect between the derivation trees and induction for me.*

If page 61 and the following section don’t help, then yes, this is probably an office-hours thing.

**What should the implementation do when the semantics doesn’t specify behavior? A checked run-time error? If so, which one? What about so-called “undefined behavior”?**

**What can we do to make the code run fast?**

I’m not sure what you mean by “expressing the leap.”

*Is there a way to represent run-time errors?*

Yes. The most common is to develop a more expansive notion of the result of a computation. Instead of a value, evaluation might result in a behavior. Examples of behaviors might include “returns 7”, “throws exception overflow”, or “halts with an assertion failure.”

*Rules seem to involve going through and finding one suitable—how to allow the interpreter to sort through quicker?*

Case statements, which in C/C++ are called “switch statements.” These allow you to find the syntactic form in constant time, and given a particular syntactic form, there are typically at most one or two rules, so you can find one there also in constant time.

*How would you go about coming up with a way to translate from one language to another given the semantics of each language? Specifically, I am thinking Impcore to UMasm, from COMP 40.*

This is a bigger topic than I can answer in a small space. But you start by considering the representation of each part of the source language semantics. How will the UM represent values? How will the UM represent environments? Next you check that every rule in the source is correctly implemented by one or more rules in the target. For example, you’ll have to figure out how to implement \( if \) using \( goto \) and conditional branches. As a semanticist or compiler writer, ultimately you are thinking about representing the source-language semantic state in the state of the target machine.

**Environments and their notation**

*What is an environment? Is it wrong to think of environments as being passed functions as well as expressions?*

An environment is a data structure or partial function that says what every name stands for. It is wrong to think of environments as being passed either a function or an expression. An environment is passed only names (to find out what the name stands for).

*How can you show that an environment actually exists? Do we assume they exist when writing the rules?*

When we’re serious about proving facts, we often prove that they hold in the empty environment, or in any arbitrary en-
environment, both of which exist. When writing rules, we just assume they exist, which is tantamount to assuming that some functions and variables are defined. We might show the assumption is good by actually showing the definitions.

Can you go from $\xi$ to $\xi'$ directly? Why would you?

The names are arbitrary, so technically you could do it. You would do it if you wanted to annoy or confuse people, or to ask a trick question on an exam (which I would find annoying).

Enrichment: $\xi$, $\xi'$, $\rho$, and all their friends have a similar status to programming-language variables, and in fact they are called metavariables. The names are arbitrary, but if we want anybody to read the code semantics, we’d better choose good names.

When do we use $\rho\{x \mapsto 3\}$?

When we want to say, “program state has been updated by setting the value of formal parameter $x$ to 3.”

What’s the difference between $\rho\{x \mapsto 3\}$ and $\rho(x) = 3$?

This is close to a reading-comprehension question: the first one is a new environment that is like $\rho$ except $x$ is now 3. The second one just says, “we know that in the initial state, there is a formal parameter $x$ and its initial value is 3.”

Does $\rho\{x \mapsto 10\}\{x \mapsto 17\}$ represent an environment that has changed?

In our minds, yes. It’s an initial (unspecified) environment that has been updated with two assignments. In the math, though, it’s just a mapping from variables to values. For example, in

(define f (x)
  (begin
    (set x 10)
    (set x 17)
    x))

(f 17)

at the end of the function, has $\rho$ changed? No. Variable $x$ is once again 17.

Is there ever an appropriate time to write $\xi'\{x \mapsto 4\}$ given that we know that $\xi$ has changed? (i.e. should the prime symbol be for general cases only?)

The prime symbol is always for cases where we don’t know. So if you know how $\xi$ has changed, you write $\xi\{x \mapsto 4\}$.

If a semantics rule has a result of an environment being prime, can we know for sure whether the environment is not changing even if we knew all the code?

I’m not sure I understand this question. The only way to be sure the environment is not changing is to run the code. If you know all the code, you can indeed run it.

Why do we need separate environments for formals and globals?

We don’t. Scheme puts them all in one environment.

Aren’t global variables just formals with global scope?

I’m not so sure. I feel like the words are getting in the way. Aren’t formals just global variables with local scope? Show me your semantics, and I will know what your variables are doing.

Why do we use Greek letters instead of reserved names?

We’re trying to pack an entire language design into a small space, as on page 81, so shorter names are better. Incidentally, this is the same reason that physicists use Greek letters. It also helps that, unless you read and write Greek, the Greek letters come without any preconceived associations. It is not an accident that common, familiar concepts like “expression” and “value” are abbreviated with Roman letters, and only less familiar concepts like “environment” and “store” are abbreviated with Greek letters.

The number of Greek letters in common use is small, and you will get to know them.

Incidentally, I know a couple of prominent type theorists who insist on Roman letters everywhere. I find their work hard to follow.

Function environments

Is there any rule that will change the environment $\phi$?

Yes; it’s the DEFINEFUNCTION rule on page 27, which is one of the rules for evaluating definitions.

Why do we have to include $\phi$ if there’s no possibility of a $\phi'$?

We have to include $\phi$ on the left so we know how to call a function. We don’t actually have to include $\phi$ on the right of the evaluation judgment for expressions, but we do use it in the evaluation judgment for definitions.

Could $\phi$ change at run time? What would this mean in the context of C—changing a function pointer?

$\phi$ could change at run time as a result of evaluating a new definition. For example, you could use the use form to load new definitions from a file, and that would change $\phi$.

In the context of C, a function pointer is firmly in the $\xi/\rho$ camp: it is a variable like any other. The only way to change $\phi$ is to load new source code.

Writing derivations

Is there a canonical or preferred way to include actual values in a proof tree? Alternatively, what is the best way to
emphasize the flow of values and changed environments in the tree?

When values don’t take much space to write, like numeric values, write the literals directly in the proof. When they take up lots of space, write names like \( v_1 \) and say in an equation off to the side what \( v_1 \) is equal to.

Why are the derivation trees not shaped in a way where they are easier to write down? I get the advantage of looking at a tree after it’s complete, but the creation is painful.

I don’t think anybody knows an easy way to write a derivation tree by hand, or to explain one for that matter. Two typical solutions are (a) to write a computer program to create the tree, or (b) to write the derivation in the style of a high-school geometry proof, with numbered sub-judgments and applications of each rule. (Do high-school students still write such proofs?)

In derivation trees, are the premises of one subbranch the conclusions of the one above it? This has been causing me all kinds of confusion.

I’m sorry you feel confused, because you have it exactly right: the premises of the root node are the conclusions of the subderivation above it.

Why don’t we write derivation trees more vertically?

If we were to write the trees more vertically, it would be harder to confirm that the derivation is valid—that is, to confirm that every premise of every rule is properly supported by a subderivation.

What is the root of a derivation?

The rule used to prove a judgment about the expression being evaluated.

In latex, often two long statements that should be next to each other, like this

```
  -- A -- B
  ----------- C
```

wind up stacked, like this

```
  ----- B
  ----- A
  ------- C
```

How can I fix this? Can I just leave it?

No! Don’t do this. We can’t read the results. Try a smaller font or a wider \texttt{parbox}, or else name your subderivations and break them out into separate displays.

Writing good derivations

Say in an evaluation judgment you have \( \rho \) as the initial state and \( \rho'\{x\mapsto v\} \) on the right-hand side. If you want to write \( \rho'\{x\mapsto v\} \) later on in the proof, do you shorten it to \( \rho' \) or \( \rho'' \)?

You don’t shorten it to \( \rho' \), because \( \rho' \) and \( \rho'\{x\mapsto v\} \) might be different. If you get tired of writing \( \rho'\{x\mapsto v\} \), you might choose to give \( \rho'\{x\mapsto v\} \) its own name, and \( \rho'' = \rho'\{x\mapsto v\} \) is as good as any.

How do I know when the sub-derivation is done? Do I need to go down to LITERAL on every Tony Hawk peak?

Close. A subderivation is done when there are no evaluation judgments above the line. That means LITERAL, FORMAL-VAR, and GLOBALVAR.

What makes a good proof? Is there a systematic way or an S.O.P. for constructing a derivation tree?

A derivation is valid if every node in the derivation tree can be obtained by substituting for metavariables in the given rule. In other words, we are applying a rule correctly if we can say “what is \( e \) here, what is \( \xi \), what are all the things,” and the answer to that question is consistent at each point.

The standard operating procedure is the algorithm sketched in the lecture notes:

Want to solve \( \langle e, \xi, \phi, \rho \rangle \Downarrow \)?

1. Find rule I can use to prove it
   - Syntactic form of \( e \) narrows down to a few choices (usually 1 or 2)
   - Look for form in conclusion

2. Now check premises

3. When premise is evaluation judgment, build subderivation recursively

When should a derivation be broken into multiple trees? vs. multiple branches?

There’s one tree per expression being evaluated. If that expression is a subexpression of a larger expression, then its tree is a subtree.

For FORMAL ASSIGN/GLOBAL ASSIGN, is the \( x \) in \( \text{SET}(x, e) \) a variable \( x \) such that it needs verification that it relies on the global/formal var rules, or does \( x \in \rho \) or \( x \notin \rho, x \in \xi \) in the premises of formal/global assign suffice?

If I understand the question, the side conditions you list suffice.

More notation

When do we write VAR(\( x \)) vs. \( x \), LITERAL(3) vs 3, etc.?

We write the longer forms when we’re writing abstract syntax, and the shorter forms when we are writing concrete
syntax. When we’re confident we understand the distinction, we often cheat by using the shorter, concrete-syntax notation even when the longer, abstract-syntax notation is technically called for.

**User-defined functions**

*What even is APPLYUser and what does it do?*

It’s the rule for applying (you might say “calling”) a user-defined function (as opposed to a primitive function like +).

*For user-defined functions (page 25) what is the difference between \( \rho' \) and \( \rho_n \)? They look the same to me.*

They go with two different functions. Let’s say I’m calling function \( f \). Then \( \rho' \) are \( f \)’s formal parameters when it’s about to return. \( \rho_n \) are my own formal parameters just after the call returns.

*How would you say in English the APPLYUser rule?*

At length, here’s how I paraphrase it (page 25):

1. I’m applying \( f \) to a list of \( n \) actual parameters
2. \( f \) has to be a user-defined function with \( n \) formal parameters. The number of formals matches the number of actuals, and each formal parameter has its own unique name.
3. I evaluate each of the actual parameters in order, producing a list of \( n \) values. Evaluation leaves the global variables in state \( \xi_n \) and my own formal parameters in state \( \rho_n \).
4. I allocate a fresh frame on the call stack, setting the initial values of \( f \)’s formal parameters to the values I got for the actual parameters. Then I evaluate the body of \( f \) to get \( v \). Evaluating the body might change global variables, producing \( \xi' \).
5. I pop the call stack, abandoning \( f \)’s formal parameters, and the result of the call is \( v \). All changes to global variables are preserved, as are any changes to my formal parameters that occurred.

**Primitive functions**

*Primitives vs. Predefined: Are primitive rules an abbreviation of the apply rule and why are they given separate judgment rules?*

Each primitive rule is its own rule and is not an abbreviation of anything else.

Primitive functions need their own rules because a primitive function doesn’t have a body—so there has to be some other way of saying what it does. Writing a lot of rules is tedious, but it keeps things simple. (In Scheme, we’ll specify some primitives using algebraic laws instead of rules.)

*Can both primitive and user defined opsem be used as root?*

I’m not sure I understand the question. But yes, both the application of a primitive function, as in \((+ 2 2)\) and the application of a user-defined function, as in \((\text{mod} 2 2)\), can be at the root of a valid derivation.

**New language designs**

*How are operational semantics designed?*

You have some idea what you think is happening at run time (or in a brand new language, what you want to happen at run time). Then you use operational semantics to write it down. The creative design generally happens in the language itself, and the semantics gives you a vocabulary in which you can say exactly what the creative work has produced.

*What would it look like to create all of the rules for a new language? What is the process like?*

You start not with the rules but with the judgment forms. For both judgment forms and rules, you would start with the language feature that most interests you, then build out from there. Experience helps: once you’ve seen semantics for three or four different languages, you have a lot of ideas you can reuse.

*When creating rules, how to prevent ambiguous situations?*

If by “ambiguous” you mean “nondeterministic,” have in mind where nondeterminism can arise, and write rules that eliminate it. For example, specify clearly what order things happen in, or establish that order doesn’t matter. To confirm you’ve succeeded, write a determinism proof like the one you’re doing for homework.

Incidentally, there are plenty of language situations where nondeterminism is appropriate. For example, if your program is running on multiple cores at once, you can’t totally control the order of events.

If you truly mean “ambiguous,” as in, “the semantics doesn’t say clearly what is supposed to happen,” one advantage of operational semantics is that it always says something. Semantics that a layman might call “ambiguous” usually manifest as nondeterministic. So a nondeterminism proof can be a good way to suss these out as well.

**Semantics more broadly**

*How many different kinds of semantics can be applied to the same language? How are these improved on? How are they formulated? Is this even possible?*
The three big families are operational semantics, denotational semantics, and axiomatic semantics. There is also an outlier called “action semantics,” and doubtless others I don’t know about. Operational semantics is further divided into “big-step” semantics and “small-step” semantics.

You improve a particular semantics by making it clearer and easier to express what you care about—or easier to prove theorems you care about. It’s much like improving code.

Other formulations of semantics are beyond what I can fit in a short answer.

What is the main advantage of using operational semantics compared to other semantics?

It’s easy to translate the semantics into an interpreter—sometimes even into an efficient interpreter. And proofs of safety properties are relatively easy.

Design of semantics

Is there any opsem rule that distinguishes the state of being inside a function and outside of one (global)?

Not in either of the forms of judgment I have shown you. For example, you can’t tell the difference between an expression inside a function and an expression on the right-hand side of a val definition. If you wanted that information, you’d have to change the form of judgment to include, in the initial state, the context of the evaluation.

Examples from class

I’m still a bit confused about why after evaluating an expression, we are certain the domain of the environment changes doesn’t change. In other words, how do we know an expression cannot create new global/formal variables?

If an expression could create a new variable, it would have to happen at an identifiable point in the computation: at a transition from one state to the next. That transition would have to be described by a rule. But we can prove (as sketched in class) that there is not any rule in the semantics that allows for creation of a new variable.

Can you elaborate on how an adversary can take advantage of nondeterminism if added to impcore (such as a clock or coin flip)?

The adversary can ruin your proof by using the clock or the coin flip in a counterexample: either of these constructs would make Impcore nondeterministic. (I don’t know how to use these features to ruin a program.)

Exam questions

What is an example of the kinds of OPSEM questions that we might be asked on an exam?

Here are some (relatively generic) examples:

- I show you some rules and you explain what they mean.
- I pick a language feature you have seen before, probably in another class, I write a semantics for it, and I ask you to identify and explain the feature just by reading the semantics.
- I pick a language feature you have seen before, probably in another class, and I ask you to write a semantics for it.
- I pick a language feature you haven’t seen before, probably something simple, and I ask you to write a semantics for it.

My favorite is probably the second.

Syntax and semantics

Why is empty BEGIN a rule but not, say, empty WHILE?

Because an empty while is not syntactically valid Impcore. Only good syntax gets a rule.

What is an empty BEGIN expression used for?

When you’re writing imperative code, it’s the best way to “do nothing.” Here’s a statement in C/C++, which “does nothing” on the (missing) else branch:

```c
if (n > 0) {
    n = n / 2;
}
```

To translate this statement into Impcore, which does not permit us to omit the else branch, I put an empty begin into the else branch:

```impcore
(if (> n 0)
  (set n (/ n 2))
  (begin))
```

Language design

When we talk about a language’s semantic flaws, what is a semantic flaw vs. syntax?

I don’t think I can answer this one except by example:

- Don’t like the symbols on the page: syntactic flaw (concrete syntax)
- Missing my favorite control construct, like for loop or do-while loop: syntactic flaw (abstract syntax)
• There’s no expression form that evaluates to a struct: syntactic flaw (abstract syntax)
• Semantics can’t specify what error occurs when something goes wrong: semantic flaw
• Semantics can’t account for a local variable that is shared among multiple functions: semantic flaw
• Semantics treats global variables and formal parameters as different when in a simpler semantics they could be treated the same: semantic flaw
• I want to add syntactic forms like &x and *p, but the semantics gives me no way to express what they can do: semantic flaw
• Semantics doesn’t handle input/output: semantic flaw

Why is the while the base loop: why not a for which implements the while?
The while loop has a simpler implementation and (related) a simpler semantics. Also, while loops won the great control-structure wars of the 1960s. (Search for “goto considered harmful.”)

Semantics of Impcore

When you use SET(x, e) it says that it doesn’t work unless x is already in the environment. How do you add a new variable to the environment then?

Two or three ways:
• In the body of a function, the formal parameters are added to the environment \( \rho \).
• The val definition form adds a global variable to the environment \( \xi \).
• You can extend Impcore with a locals form that adds local variables to an environment.

Does what we learned about today w.r.t. \( \text{dom} \xi = \text{dom} \xi' \) also apply to other environments?

Yes. You can prove the same theorem about \( \rho \) and \( \rho' \).

Is the list of rules on the handout fully exhaustive?

No. There are some rules in the chapter that aren’t in the summary.

What’s the difference between GlobalAssign and DefineGlobal?

GlobalAssign is written using set, and it is an expression form. DefineGlobal is written using val, and it is a definition form. You can use set inside a function, but you cannot use val there.

Data types

How would you represent data types using Impcore-like operational semantics?

Exactly the same way as in Scheme. You’ll typically find data types specified using a BNF grammar. (Since BNF is no longer taught in COMP 11, I tend not to lean on it too heavily in 105.) But any of the techniques in the first section of the handout on scheme values would also work.

Initial basis

What’s an initial basis and why is it important?

It’s all the primitive and predefined names, together with their meanings. In layman’s language, it’s the “standard library.” It’s important because it has a huge effect on usability and productivity. Python, for example, is known for the amazing breadth of its libraries.
What is the formal name for the clock you have? I keep referring to it as a ‘belt-watch’ and I’m guessing there’s a better term.

The person who gave it to me called it a “belt watch.” But if you want to find one for sale, you are better off searching for “belt clip watch.”