NTNU – Trondheim
Norwegian University of
Science and Technology

Operational semantics

(Extracurricular perspective,
there will be no quiz questions on this)

Once again, from the top

- *Lexically*, a language is just a pile of units which can't be divided without losing or altering their meaning
  - Written English is nicely partitioned by punctuation and spacing (eventhoughyoucanstillreaditwithoutthemitisjustharder)
  - Words like *tigerlily* get their own places in the dictionary
  - If we could generalize from tigers and lilies to combine arbitrary animals and flowers, it would be syntactical

    ...but it's the name of a thing, that's something else…

    ...so it gets to be its own lexical entity.

Syntax (grammar)

We ← subjects are nouns, pronouns
form ← verbs determine the predicate
sentences ← direct objects are nouns, pronouns
correctly. ← adverbials are adverbs

In English, syntactic function is mostly determined by the position of a word in a statement.

(Other langauges do it by declining the words instead, but I have yet to see a *programming* language which handles it that way)

At the edge of meaning

- Syntactically correct statements don't necessarily "mean" anything.

  "Colorless green ideas sleep furiously"

  In a way, it means that we have an illustration of a meaningless, yet syntactically correct statement, but you would never know from the statement taken out of this context.
- The grammars we've been noodling with are called *context-free*
- That's because they aren't the least bit helpful in separating statements from properly structured nonsense

Meaning is a tricky word

- "birds of a feather" ← *idiomatically*, has little to do with actual birds
- "a big heart" ← *pragmatically*, means one thing in a festive speech, and another in hospitals
-  ← *semiotically*, means that this substance will ruin your dinner
- `Even changing typefaces alters how we perceive things` →

...it's all different kinds of *meaning*….

Program semantics = program meaning

- ...for very carefully selected values of meaning
- *Semantics* connect a statement and its environment to the structure of the statement itself
- That is, for a statement like “divide x in two equal parts”
  - If x is a number, divide it by two (both halves will be equal)
  - If x is a string, chop it up in the middle
  - If x is a cake…

    ...you get the picture
- Programming language semantics usually have a lot to do with *types*, and how to organize them

Flavors of semantics

- Painting with broad strokes, we have
  - *Operational* semantics, which describe the meaning of a statement in terms of what you do to the environment in order to create its effect
  - *Denotational* semantics, which describe how the environment is affected by a statement without specifying the steps taken to make it so
  - *Axiomatic* semantics, which describe properties of the environment which are preserved throughout a statement
- This is a subject unto itself, which we shall mostly leave alone
  - It's of greater interest to the language design folks anyway
- Nevertheless, we should touch upon it
  - A compiler must respect the source language semantics
  - Otherwise, it becomes a compiler for a different language

Why I am showing you this

- Looking at what the Dragon has to say about types, one might get the impression that semantics has to do with syntax tree traversal order
  - Dragon attaches type information in the notation of attribute grammars, and puts it in semantic actions, *cf.* order restrictions of L- and S- attribution (subchapters 6.3, 6.5)
- A bit of perspective can be harvested if we take 2 steps back and look at the topic from afar
- Therefore, I feel it is appropriate to do that
- Naturally, I think that you should feel this way also.

  (We *are* taking a detour from the syllabus, though, it doesn't say anything about what I will tell you today)

A small language to look at

- This is the syntax of the *While* language*:

  a → n | x | a1 + a2 | a1 * a2 | a1 – a2

  b → true | false | a1 = a2 | a1 ≤ a2 | ¬b | b1 & b2

  S → x := a | skip | S1 ; S2

  S → if b then S1 else S2 | while b do S

- We assume a lexical specification so that
  - n is a numeral (Num)
  - x is a variable (Var)
  - a is an arithmetic expression (Aexp), with the value A[a]
  - b is a boolean expression (Bexp), with the value B[b]
  - S is a statement (Stm)

*unceremoniously borrowed from the book *"Semantics with Applications: An Appetizer"*, Nielson & Nielson (2007)

In plain English

- We need a notation to talk about how statements affect their environment

[x→1, y→2]	means “x is 1 and y is 2 in this state”
s	means a final state after some statement
< S, s >	means statement S is executed in state s
A[a] s	means the value of a in state s
B[b] s = tt	means that b is true in state s
B[b] s = ff	means that b is false in state s

Operational semantics

- This is a name for the style of semantic specifications that represent program execution using *operations* on the environment
  - You can invent your own, there isn't a final list of all the ways to take this approach
- To get a feel for what they're like, we will define the *While* language in two ultimately equivalent ways:
  - *Natural* semantics
  - *Structural operational* semantics

  (These are names for specific notations)

Natural semantics

- Natural semantics describe what a state is like before a statement, and after it
- Everything that's true about the program is written in the form $< S, s > \rightarrow s'$
- This indicates that running a (possibly compound) statement S when the system is in state s will
  - Stop at some point
  - Leave the system in state s' afterwards

A first rule

- The simplest statement we've got is one that has no effect at all:

  $< \text{skip}, s > \rightarrow s$

- That is, executing 'skip' in state s doesn't alter the state s at all
- Specifying natural semantics is a game of hocking up rules like this one for all types of statements permitted by the grammar

Assignments

$< x := a, s > \rightarrow s [x \rightarrow A[a] s]$

- Our end state (s') here is "$s [x \rightarrow A[a] s]$"
- This means that executing "x := a" in state s turns it into state s again, except that x is now bound to the value of expression a, as evaluated in state s

Applying another assignment like "y := x" afterwards gives us

$< y := x, s [x \rightarrow A[a]s] > \rightarrow s [x \rightarrow A[a]s] [y \rightarrow A[x] s[x \rightarrow A[a]s]]$

but it's more readable to write in two steps and substitute

$< x := a, s > \rightarrow s [x \rightarrow A[a] s]$ ($s' = s [x \rightarrow A[a] s]$),

$< y := x, s' > \rightarrow s' [y \rightarrow A[x] s']$ ($s'' = s' [y \rightarrow A[x] s']$),

...and so on, from s''.

Composition

- When statement S2 follows S1, we write

$$\frac{\langle S1, s\rangle \rightarrow s' \qquad \langle S2, s'\rangle \rightarrow s''}{\langle S1;S2, s\rangle \rightarrow s''}$$

  to mean that S1 goes from s to s', S2 from s' to s''
- The *immediate constituents* above the line are our *premises*, what's below is the *conclusion*
- Skip and assignment rules don't have any premises, that makes them *axioms*

If in two flavors

- If, when it is taken, (*i.e.* when B[b]s=tt) says that

  $$\frac{\langle S1, s\rangle \rightarrow s'}{\langle \text{if b then S1 else S2}, s\rangle \rightarrow s'}$$

- If, when it is **not** taken, (*i.e.* when B[b]s=ff) says that

  $$\frac{\langle S2, s\rangle \rightarrow s'}{\langle \text{if b then S1 else S2}, s\rangle \rightarrow s'}$$

- All it means is that S1 modifies s when b is true, and that S2 gets to do it when b is false

  (it really is just a strict notation to summarize how we intuitively want if statements to work)

While in two flavors

- When B[b]=tt

$$\frac{\langle S, s\rangle \rightarrow s' \qquad \langle \text{while } b \text{ do } S, s'\rangle \rightarrow s''}{\langle \text{while } b \text{ do } S, s\rangle \rightarrow s''}$$

  and when B[b]=ff

  $\langle \text{while } b \text{ do } S, s\rangle \rightarrow s$

- That is, when the condition is true, S runs once more
- When the condition is false, while is just like skip

The semantic function S_{NS}

- This whole specification lets us math-ify the effect of any statement S in the language, to think of it as a partial function from state to state:

  S_{NS} [S] = s' when <S,s> → s'

  S_{NS} [S] = undef otherwise

Structural operational semantics

- We can also specify semantics in a more detailed, explicit step-by-step manner
- Let the relation => be either

  between two configurations <S,s> => <S',s'>

  from a configuration to a state <S,s> => s'
- The idea is to write out not just what statements do to a state in the end, but also all the steps taken in order to make it so

Skip is still easy

<skip, s> => s

- As before, this doesn't do anything by design

Assignment

<x:=a, s> => s[x → A[a]s]

- This looks pretty much the same as well, an assignment gives a value to a variable

Composition 1

$$\frac{\langle S1, s\rangle \Rightarrow \langle S1', s'\rangle}{\langle S1;S2, s\rangle \Rightarrow \langle S1';S2, s'\rangle}$$

- This is because we haven't assumed that (compound) statements run to completion, so their intermediate work must be represented
- Here, S1 is not finished yet

Composition 2

$$\frac{\langle S1, s\rangle \Rightarrow s'}{\langle S1;S2, s\rangle \Rightarrow \langle S2, s'\rangle}$$

- Here, S1 completes, resulting in a state which S2 can get to work on

If in two flavors

<if b then S1 else S2, s> => <S1, s> when B[b]s=tt

<if b then S1 else S2, s> => <S2, s> when B[b]s=ff

- These look a little more like how programmers mentally run programs in their minds
- i.e. “S1 runs when condition is true, S2 runs otherwise”

While

<while b do S, s> =>

<if b then (S; while b do S) else skip, s>

- This is an equivalence which makes 1 (more) iteration explicit in terms of condition and loop body

  (and leaves the next iteration to be evaluated when we get there)

The semantic function $S_{SOS}[S]$

- As before, we can look at this as a computable function between configurations/states, parametric in the statement
- For all statements S in the language,

  $S_{SOS}[S] = s$ when S =>* s completes

  $S_{SOS}[S] = undef$ otherwise

$S_{NS}[S] = S_{SOS}[S]$

- These two specifications give the exact same interpretation of *While* programs
- Maybe you can see it, we won't spend time on proving it
- One of the things that says, is that the N.S. notation can be elaborated into every tiny sub-step, it's not missing any detail
- It's much shorter to write, so that will do

This is some kind of logic

- *Natural deduction* is a formalism like this which was invented before digital computers, in an inter-war era attempt to write formal rules for common-sense reasoning
  - The sort of thing philosophers like to do
- Natural semantics emerges when you try to use such rules specifically to describe what a computer program does, it turns into states and steps

Looks like an intricate way to write the obvious

- Fundamental things do, once they're discovered
- If we pretend that high and low voltages represent 1 and 0, arithmetic and processing circuits are the same thing
  - We didn't know before Claude Shannon noticed it
- If we represent state changes in axiomatic logic, running a program and constructing a proof is the same thing
  - We didn't know before Haskell Curry and William Howard noticed it

But what's the *point*?

- If you have a program and its proof, all it really means is that you wrote the same thing twice:
  - Once in source code
  - Once with pen and paper
- The point is to detach what a program is supposed to do from what the compiler's translation actually does
  - If you have only the compiler's interpretation, that's what defines the meaning of the program
  - If you have another definition handy, you can find it out if the compiler makes mistakes

Language specifications

- As you may have noticed, working out appropriate rules and proof trees takes a lot of effort even for a tiny toy language
- With a few notable exceptions, language standards tend to be explained by thick books in English instead, as unambiguously as the standard committee manages to write it
- Formal definitions tend to be reserved for limited situations where mistakes are incredibly expensive

Sidebar:

Our language specification

- If you haven't noticed yet, you will: the VSL language we implement in the exercises is neither completely nor formally specified
- I don't think that's actually a bad thing:
  - Even if I wrote a 100 pages of illegible definitions, I would probably still miss some subtle point lurking in a corner
  - When you find something under-specified, it makes you consider what you think the language should mean, and how to implement it

...no, ***really*** - what's the point?

- Oh, all right.
- If you take the states and their changes out of the notation, you're left with the logical inferences you can make about program semantics based only on what the source text says
- That's what a static type system does
- I like today's notation better than the book's way to write type checking rules, so I plan to use it for that.

  (It is, in fact, equivalent, just wanted to introduce it)

The Great Perspective™

- If we disregard the notation thing, there are still two great ideas here:

  1) There are ways to show that a systematic way to change a program can leave its meaning alone

  (*Semantics-preserving transformation* is what an optimizer does)

  2) You can think of a block of statements as a way to write down its *semantic function*

- Both of these ideas can bring order to the understanding of dataflow analysis, at the end of the semester
- You can understand dataflow analysis without them too, I just think they're a lot easier to grasp when you meet them for the 2nd time, after leaving them alone a while