CPSC 311: Definition of Programming Languages: Evaluation semantics: More functions ("08-morefunctions")

Joshua Dunfield
University of British Columbia

October 4, 2015

Topics discussed

- interpreter for Fun (**Racket code**: `dynsem-fun.rkt`)
- first-class functions
  - mathematicians also think they're weird
  - functions are values...
  - ...but are displayed in an unhelpful way by most language implementations
  - when are functions equal?
- unparsing
- recursion:
  - evaluation rule for "rec"
  - "derivation tree" really means "*finite* derivation tree"
  - we can write a "base case", *or* a "recursive case", but useful recursive functions have both base and recursive cases; is this possible in Fun?

    (**the lecture on Friday, 2015/09/25 ended around this point**)
  - let's make it possible in a *reasonable* way: add ifzero
  - syntactic sugar
  - soundness and completeness; error handling; examples of undefined behaviour

    (**the lecture on Monday, 2015/09/28 ended around this point**)

1 Collected rules for Fun

Figure 1 collects all the rules, showing Eval-app-value rather than Eval-app-expr.

2 From the Fun rules to a Fun interpreter

For AEs and WAEs, we said that an interpreter should do this:

"Given an ae, find a number n such that $ae \Downarrow n$."

$$\frac{}{(\text{num } n) \Downarrow (\text{num } n)}\ \text{Eval-num} \qquad \frac{e1 \Downarrow (\text{num } n_1) \qquad e2 \Downarrow (\text{num } n_2)}{(\text{add } e1\ e2) \Downarrow (\text{num } (n_1 + n_2))}\ \text{Eval-add} \qquad \frac{e1 \Downarrow (\text{num } n_1) \qquad e2 \Downarrow (\text{num } n_2)}{(\text{sub } e1\ e2) \Downarrow (\text{num } (n_1 - n_2))}\ \text{Eval-sub}$$

$$\frac{e1 \Downarrow v1 \qquad subst(e2, x, v1) \Downarrow v2}{(\text{with } x\ e1\ e2) \Downarrow v2}\ \text{Eval-with} \qquad \frac{}{(\text{id } x) \Downarrow \text{free-variable-error}}\ \text{Eval-free-identifier}$$

$$\frac{}{(\text{lam } x\ e1) \Downarrow (\text{lam } x\ e1)}\ \text{Eval-lam} \qquad \frac{e1 \Downarrow (\text{lam } x\ eB) \qquad e2 \Downarrow v2 \qquad subst(eB, x, v2) \Downarrow v}{(\text{app } e1\ e2) \Downarrow v}\ \text{Eval-app-value}$$

Figure 1 Evaluation rules for Fun

For Fun, we have a more general idea of the result of evaluation that includes functions as well as numbers, and we said that functions (lam x *e*) and numbers (num n) are collectively *values*, so a Fun interpreter should do this instead:

"Given an *e*, find a value *v* such that $e \Downarrow v$."

(During lecture, we updated the `interp` function in `dynsem-fun.rkt`.)

3 Fly first-class, for free

If we wrote our interpreter correctly, we now have a programming language that is quite similar to the λ-calculus (which was invented by Alonzo Church in the 1930s, and extensively studied ever since). As a programming language, the λ-calculus has very few features (it doesn't even do arithmetic. . . at least not in a way that you'd recognize), but it does have functions that are *first-class*—functions that can take other functions as arguments, and return functions. You may not have noticed it, but our rules have no trouble with first-class functions.

3.1 It's not just you

First-class functions are often considered a strange and advanced language feature. Back in 1967, Christopher Strachey (who studied the semantics of programming languages, and also—rather curiously—set in motion a chain of events that led to C) pointed out that mathematicians rarely treated functions as "first-class" and hadn't even agreed on a notation for first-class functions; mathematicians seemed to have little grasp of how to use functions as values.

Today, the DrRacket "Beginning Student" language doesn't allow functions as arguments, and it doesn't allow a function to return a function. The "Intermediate Student" language allows functions as arguments, but not as results. The "Intermediate Student with `lambda`" language adds the ability to return a function (by returning a `lambda`).

This distinction may be useful when learning how to program; I'm not sure it's useful when learning how to think about defining programming languages. **Functions are values;** until they are applied, they don't do anything, just as numbers don't do anything until you do arithmetic on them.

Unfortunately, most "real" programming languages make that hard to remember. For example, DrRacket claims that the result of evaluating (**lambda** `(x) x`) is "`#<procedure>`". But that's a bad, secretive notation for the function you entered; you should think of it as being (**lambda** `(x) x`). Two other functional languages, SML and OCaml, also refuse to show you the inside of a function.

```
Standard ML of New Jersey v110.72 [built: Tue Jan 11 13:30:58 2011]
- 2 + 2;
val it = 4 : int
- (fn x => x + x) 2;
val it = 4 : int
- (fn x => x + x);
val it = fn : int -> int
```

```
        OCaml version 4.00.1

# 2 + 2;;
- : int = 4
# (fun x -> x + x) 2;;
- : int = 4
# (fun x -> x + x);;
- : int -> int = <fun>
```

Two different Haskell implementations, hugs and ghci, not only won't show it, but print mysterious error messages, just to taunt you.

```
__   __ __  __  ____   ___      ------------------------------------------
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2005
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Bugs: http://hackage.haskell.org/trac/hugs
||   || Version: September 2006 ------------------------------------------

Hugs> 2 + 2
4
Hugs> (\x -> x + x) 2
4
Hugs> (\x -> x + x)
ERROR - Cannot find "show" function for:
*** Expression : \x -> x + x
*** Of type    : Integer -> Integer
```

```
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Prelude> 2 + 2
4
Prelude> (\x -> x + x) 2
4
Prelude> (\x -> x + x)

<interactive>:4:1:
    No instance for (Show (a0 -> a0)) arising from a use of ~print~
    In a stmt of an interactive GHCi command: print it
Prelude>
```

Python displays the function along with its address in memory:

```
Python 2.7.2 (default, Oct 11 2012, 20:14:37)
>>> 2 + 2
```

```
4
>>> (lambda x: x + x) (2)
4
>>> (lambda x: x + x)
<function <lambda> at 0x1023d3938>
```

Why is this? I'm not sure. It kind of makes sense for a compiler to throw away abstract syntax, but Hugs isn't even a compiler. The point of a REPL (read-eval-print loop; also known in DrRacket as "Interactions", and in many other languages as a "toplevel") is not to be efficient, but to allow "playing" with a language by typing in expressions and seeing what happens. It doesn't seem that difficult for something like SML to preserve the abstract syntax of code, at least code that you enter in the REPL. But I haven't implemented it myself, so there are probably issues I haven't thought of.

(Apparently, JavaScript *will* show you the actual function definition! I don't like JavaScript, but that's definitely a point in its favour.)

3.2 Looking behind the curtain

The distinction between first-class functions and "lesser" functions may matter, however, when we try to write *efficient* interpreters and compilers. But not worrying about efficiency is helpful now: Because our interpreter follows a (relatively) very simple evaluation semantics, you can get an idea of how first-class functions work *in general* by writing Fun expressions that evaluate to functions, and *looking at the functions*—our Fun language always shows you what's inside a lam! Then you can take that *general* idea and use it when programming in Racket, OCaml, or Haskell.

This may not be too effective yet, because our Fun language is so small, but you'll add several features to it in the next assignment.

3.3 Unparsing

A parse function takes concrete syntax (for us, an S-expression) and builds abstract syntax. An "unparse" function takes the abstract syntax, and turns it back into concrete syntax.

I'm doing this now so that when you look at the lams your Fun expressions evaluate to, you can read them more easily.

For convenience, I'm using quasiquote and unquote. (A fun (?) exercise: write a version of `unparse` that *doesn't* use quasiquote and unquote.) The file `dynsem-fun.rkt` has some useful links about these features.

3.4 Digression: equality of functions

When are functions equal? Not an easy question!

In Racket, after defining a function using **define**, that function is equal to itself. But if we write identical expressions using **lambda**, they are not equal.

```
> (equal? unparse unparse)
#t
> (equal? (lambda (x) x) (lambda (x) x))
#f
```

Python works similarly.

SML, rather characteristically, just refuses to let you compare functions at all:

```
- (fn x => x) = (fn x => x);
stdIn:1.1-1.26 Error: operator and operand don't agree [equality type required]
  operator domain: ''Z * ''Z
  operand:         ('Y -> 'Y) * ('X -> 'X)
  in expression:
    (fn x => x) = (fn x => x)
```

Its complaint is that functions don't have "equality type"; they don't have a type for which SML defines equality. The principle here is that the answer to whether two functions are equal is most likely useless—it would be reasonable to expect that `(lambda (x) x)` would be equal to itself, but it's not, so SML just doesn't define equality on functions at all.

OCaml defines two (at least) kinds of equality: `=` doesn't work for functions (with an exception, rather than a type error), and `==` works like `equal?` in Racket: it *might* return `true` for functions that are the same, but it might just return false.

```
# (fun x -> x) = (fun x -> x);;
Exception: Invalid_argument "equal: functional value".
# (fun x -> x) == (fun x -> x);;
- : bool = false
# let identity = fun x -> x;;
val identity : 'a -> 'a = <fun>
# identity == identity;;
- : bool = true
```

Neither approach to function equality (not defining it at all, or defining an equality test that often says "no, they're not equal" even for functions with identical source code) is totally satisfying. A mathematician's answer to the question, "Are the function $f(x) = x + 1$ and the function $g(y) = y + 2 - 1$ the same?" would be (I think—I'm not really a mathematician) "yes", because both functions are *extensionally equal*: given the same arguments, they produce the same results.

In general, the question of whether two functions are extensionally equal is undecidable, and certainly difficult: imagine that the bodies of the functions `f` and `g` above were each 100,000 lines long. An interesting approach would be to implement a version of function equality that has *three* possible answers: "these functions are obviously extensionally equal", "these functions are obviously extensionally not equal", and "I don't know". I don't know if anyone has tried this approach.

4 Recursion

A reasonable objection to our Fun language so far is that we can't write recursive functions, so let's address that.

The approach we'll take is not to add recursive *functions* as such, but a recursion *expression* whose body can be a function. For example, we can write (`rec` u (`lam` x *e*)), where *e* is some expression that can refer to u and x.

(It would be slightly more standard to write "`fix`" instead of `rec`, for "fixed point", but we won't concern ourselves with whatever a fixed point might be. I mention this only to encourage you to yell at me if I write `fix` by accident.)

$$\begin{aligned}\langle \mathrm{E} \rangle \ ::= \ & \vdots \\ & \mid \{\texttt{rec}\ \langle \mathrm{id} \rangle\ \langle \mathrm{E} \rangle\}\end{aligned}$$

What does this thing mean? To answer (?) that, we need an evaluation rule.

$$\frac{\mathit{subst}(e, u, (\mathsf{rec}\ u\ e)) \Downarrow v}{(\mathsf{rec}\ u\ e) \Downarrow v}\ \text{Eval-rec}$$

The identifier u in $(\mathsf{rec}\ u\ e)$ is a way for the expression e to refer to *itself*. So, to evaluate $(\mathsf{rec}\ u\ e)$, we replace u with... $(\mathsf{rec}\ u\ e)$! Unfortunately, this can lead to trouble...

A very simple example of a rec is the expression

$$\big(\mathsf{rec}\ u\ (\mathsf{lam}\ x\ (\mathsf{id}\ x))\big)$$

Evaluating this is no trouble, but the rec doesn't really serve any purpose here: u doesn't appear in $(\mathsf{lam}\ x\ (\mathsf{id}\ x))$, so substituting for u has no effect:

$$\frac{\mathit{subst}((\mathsf{lam}\ x\ (\mathsf{id}\ x)), u, \big(\mathsf{rec}\ u\ (\mathsf{lam}\ x\ (\mathsf{id}\ x))\big)) \Downarrow v}{\big(\mathsf{rec}\ u\ (\mathsf{lam}\ x\ (\mathsf{id}\ x))\big) \Downarrow v}\ \text{Eval-rec}$$

Using the definition of *subst*, the premise is really

$$\frac{(\mathsf{lam}\ x\ (\mathsf{id}\ x)) \Downarrow v}{\big(\mathsf{rec}\ u\ (\mathsf{lam}\ x\ (\mathsf{id}\ x))\big) \Downarrow v}\ \text{Eval-rec}$$

Using Eval-lam, we get

$$\frac{\dfrac{}{(\mathsf{lam}\ x\ (\mathsf{id}\ x)) \Downarrow (\mathsf{lam}\ x\ (\mathsf{id}\ x))}\ \text{Eval-lam}}{\big(\mathsf{rec}\ u\ (\mathsf{lam}\ x\ (\mathsf{id}\ x))\big) \Downarrow (\mathsf{lam}\ x\ (\mathsf{id}\ x))}\ \text{Eval-rec}$$

This is a perfectly good derivation, but we could have obtained the same value by omitting the rec and just writing $(\mathsf{lam}\ x\ (\mathsf{id}\ x))$.

Let's try to evaluate the simplest possible rec expression that *does* use u.

$$\frac{\mathit{subst}((\mathsf{id}\ u), u, \big(\mathsf{rec}\ u\ (\mathsf{id}\ u)\big)) \Downarrow v}{\big(\mathsf{rec}\ u\ (\mathsf{id}\ u)\big) \Downarrow v}\ \text{Eval-rec}$$

Rewriting our goal (the premise of Eval-rec) using the definition of *subst*, we get

$$\frac{\big(\mathsf{rec}\ u\ (\mathsf{id}\ u)\big) \Downarrow v}{\big(\mathsf{rec}\ u\ (\mathsf{id}\ u)\big) \Downarrow v}\ \text{Eval-rec}$$

So now we need to derive $\big(\mathsf{rec}\ u\ (\mathsf{id}\ u)\big) \Downarrow v$. The only rule that could possibly work is Eval-rec:

$$\frac{\dfrac{\mathit{subst}((\mathsf{id}\ u), u, \big(\mathsf{rec}\ u\ (\mathsf{id}\ u)\big)) \Downarrow v}{\big(\mathsf{rec}\ u\ (\mathsf{id}\ u)\big) \Downarrow v}\ \text{Eval-rec}}{\big(\mathsf{rec}\ u\ (\mathsf{id}\ u)\big) \Downarrow v}\ \text{Eval-rec}$$

This new goal uses *subst*, so we follow that definition again...

$$\frac{\dfrac{\big(\mathsf{rec}\ u\ (\mathsf{id}\ u)\big) \Downarrow v}{\big(\mathsf{rec}\ u\ (\mathsf{id}\ u)\big) \Downarrow v}\ \text{Eval-rec}}{\big(\mathsf{rec}\ u\ (\mathsf{id}\ u)\big) \Downarrow v}\ \text{Eval-rec}$$

This isn't going anywhere!

We should clarify our idea of what a derivation is: a derivation *must be finite*. Endlessly applying the same rule to get an infinite tree isn't allowed.

4.1 Base and recursive cases?

Our first attempt to use rec didn't really do anything; we wrote rec but didn't use it, kind of like the base case of a recursive function. Our second attempt had us endlessly trying to derive the same thing (and an interpreter following the Eval-rec rule would, in fact, run forever).

A third idea:

$$(\text{rec } u\ (\text{lam } x\ (\text{add } (\text{id } x)\ (\text{app } (\text{id } u)\ (\text{id } x)))))$$

This does use u. Evaluating this expression is fine—it evaluates to a lam. However, when we apply that lam to something, we'll try to evaluate forever again (though with an ever-changing goal).

Can we have both a base *and* a recursive case in one function?

Yes, but you have to `Vftnvf0dzvchd xccwdrzj rj opc0rnttqdzo ytzvosczj rzh ofdz rkkwi ofdq szjodrh cy tjszt sy0ofdz0dwjd`—I mean, use Church encodings, which are pretty unpleasant. So we're going to do it an easier way.

4.2 Conditional expressions

We need a way for a Fun expression to *test* a value, and evaluate one of two expressions depending on what the value is. So we'll add "`ifzero`".

$$\begin{aligned}\langle \text{E} \rangle\ ::= &\ \vdots \\ \mid &\ \{\text{ifzero } \langle \text{E} \rangle\ \langle \text{E} \rangle\ \langle \text{E} \rangle\}\end{aligned}$$

```
(define-type E
  [num (n number?)]
  [add (lhs E?) (rhs E?)]
  [sub (lhs E?) (rhs E?)]
  [with (name symbol?) (named-expr E?) (body E?)]
  [id (name symbol?)]
  [lam (name symbol?) (body E?)]
  [app (function E?) (argument E?)]
   [ifzero (scrutinee E?) (zero-branch E?) (nonzero-branch E?)]
)
```

With function application app, we saw that we could use either the value strategy, or the expression strategy, and each had advantages and disadvantages. For ifzero, the first step is to evaluate the "scrutinee" (because ifzero is inspecting, or "scrutinizing", this expression, to see if evaluates to zero). But should we evaluate both branches, or just one?

We want a way of writing both a base case and a recursive case—if we use the recursion variable u inside one of the branches, and we evaluate that branch, we're liable to recurse forever. So we had better not evaluate both branches! Instead, we should use these rules:

$$\frac{e \Downarrow (\text{num } 0) \qquad eZ \Downarrow v}{(\text{ifzero } e\ eZ\ eNZ) \Downarrow v}\ \text{Eval-ifzero-zero} \qquad \frac{e \Downarrow (\text{num } n) \qquad eNZ \Downarrow v}{(\text{ifzero } e\ eZ\ eNZ) \Downarrow v}\ \text{Eval-ifzero-nonzero}$$

Or should we? Something is missing.

What's missing is a premise in rule Eval-ifzero-nonzero saying that $n \neq 0$. Without this premise, when $e \Downarrow (\text{num } 0)$, we could apply either rule. That's really bad if we're trying to use ifzero to prevent unbounded recursion. It also violates determinism, that is:

"For all expressions e, if $e \Downarrow v_1$ and $e \Downarrow v_2$, then $v_1 = v_2$."

There are good reasons to violate determinism (can you think of any?), but forgetting a premise isn't one of them.

5 Syntactic sugar

This ifzero expression doesn't seem too versatile; what if we want to test if a number is *less* than zero? Should we add another kind of expression, iflessthanzero? We could, but a better, more general design is to add booleans and `if` to the language, which will be part of the next assignment.

Exercise 1. Write a Fun expression that behaves like iflessthanzero, using only ifzero and the other features of Fun (including recursion). It only needs to work for integers; don't worry about other numbers. (I think I have a solution, but I haven't written it down... it has a peculiar Turing-machine flavour.)

Less perversely, we can code up ifequal: instead of (ifequal $e1$ $e2$ $e\text{Eq}$ $e\text{NotEq}$),
write (ifzero (sub $e1$ $e2$) $e\text{Eq}$ $e\text{NotEq}$). It would be annoying to actually write that instead of ifequal. On the other hand, it would be annoying to add ifequal to the language: we would have to update our parser, add evaluation rules, extend the definition of substitution, and add code to our interpreter. (If we were proving things in 311, we would also want to extend our proofs of whatever language properties we care about, such as determinism.)

Thus, a common practice in language design is a third option: add a new feature as *syntactic sugar*. We still have to update our parser, but *nothing else has to change*, because we will translate ("desugar") ifequal within the parser:

{ifequal $e1$ $e2$ $e\text{Eq}$ $e\text{NotEq}$} is parsed as (ifzero (sub $e1$ $e2$) $e\text{Eq}$ $e\text{NotEq}$)

This seems to save a lot of work; is there any reason not to do this?

Unfortunately, yes: parsing and unparsing are no longer inverse operations. That is, transforming concrete syntax to abstract syntax (parsing) and then transforming the abstract syntax back to concrete syntax (unparsing) won't necessarily give the original concrete syntax back.

That might not sound too bad... except that unparsing is also how we would want to print error messages. So the error messages will be confusing, because they refer to code the user didn't write! For example, our interpreter should print an error message if you try to use ifequal with lams—and indeed it will, assuming that subtracting lams prints an error message. But the error message will say that sub was given invalid arguments, not that ifequal was!

Here's an example from a real programming language, SML (my favourite language). To make any sense of this, you probably need to know that `case` is SML's version of **type-case** and that `SOME` is a variant (constructor) declared by (SML's version of) **define-type**; I'll try to explain the rest as we go.

```
Standard ML of New Jersey v110.72 [built: Tue Jan 11 13:30:58 2011]
- fun f x = case x of SOME y => y
                    | SOME z => z;
stdIn:1.14-2.36 Error: match redundant and nonexhaustive
          SOME y => ...
    -->   SOME z => ...
```

SML is complaining that I've written the same constructor twice in two branches ("redundant") and also that I didn't write another constructor at all ("nonexhaustive"), errors you've already seen (with different terminology) with **type-case**.

In SML (and in PLAI), it's common to write a function that immediately does a `case` (**type-case**), so SML allows you to write functions in "clausal form", like this:

```
fun fib 0 = 0
  | fib 1 = 1
  | fib n = fib (n-2) + fib (n-1);
```

This closely resembles mathematical notation for defining functions by cases, but it behaves exactly like

```
fun fib x = case x of 0 => 0
                    | 1 => 1
                    | n => fib (n-2) + fib (n-1);
```

The Definition of Standard ML defines clausal form to be a "derived form", which is a fancy name for syntactic sugar: the meaning of a clausal function is given by a translation to a function whose body is a `case`. That is, the clausal form syntax is derived from the "real" syntax (`case`).

Since the error message shows the unparsing of the abstract syntax, it shows code that doesn't match what I wrote:

```
- fun f (SOME y) = y
    | f (SOME z) = z;
= stdIn:1.9-3.23 Error: match redundant and nonexhaustive
          SOME y => ...
    -->   SOME z => ...
```

Showing the "wrong" code teaches SML programmers which language features are derived forms, which is somewhat useful but probably doesn't make up for the frustration.

6 Soundness and completeness

What does "following the rules" really mean?

Definition 2. Completeness of the interpreter: If $e \Downarrow v$ is derivable then `(interp e)` $= v$.

If completeness does not hold, we say the interpreter is *incomplete*. For example, you might forget to implement an evaluation rule.

Definition 3. Soundness of the interpreter: If `(interp e)` $= v$ then $e \Downarrow v$ is derivable.

If soundness does not hold, we say the interpreter is *unsound*.

The words "is derivable" are not quite necessary, but I included them to emphasize that the definition of $e \Downarrow v$ is given by rules.

6.1 Bonus rant

(Skipped during lecture; feel free to skip it here too.) In "interpreter semantics", the *interpreter itself* defines what $e \Downarrow v$ means. So the definitions of soundness and completeness collapse: "If $e \Downarrow v$ then

$e \Downarrow v$."

Suppose two of you are (separately) implementing interpreters. One of you implements function application in a way that corresponds to Eval-app-value, and the other implements function application in a way that corresponds to Eval-app-expr. Under interpreter semantics, you have *both* implemented a "correct" interpreter, because *the act of writing an interpreter* (according to interpreter semantics) defines what the language is.

Interpreter semantics has another drawback: you cannot construct a language definition in which behaviour is undefined, because whatever your interpreter happens to do *is* the definition of the language. Now, *which* behaviours should be left undefined can be debated, but real programming languages have multiple implementations (even if we're only counting patches and bug fixes to a single "canonical" implementation!) and run in different environments and processor architectures; you usually can't define everything.

6.2 Undefined behaviour

To be sound, your interpreter must not evaluate an expression successfully (that is, return a value) unless the rules say it does. So your interpreter must not return a value for the expression

$$(\mathsf{add}\ (\mathsf{lam}\ \cdots)\ (\mathsf{lam}\ \cdots))$$

unless $(\mathsf{add}\ (\mathsf{lam}\ \cdots)\ (\mathsf{lam}\ \cdots)) \Downarrow v$ is derivable according to the rules (which it's not for the languages Fun and Fun++ that we've discussed).

For free identifiers, we wrote a rule Eval-free-identifier that says that evaluating $(\mathsf{id}\ x)$ is a "free-variable-error". But we haven't written rules for other "errors", like adding two lams. Thus, your interpreter can't return a value, but it's free to treat adding two lams any way you like. You could:

- generate an error (similar to free-variable-error);
- loop forever (which sounds kind of silly, but we already loop for (rec u (id u))...);
- something else entirely.

(Viktor Vafeiadis, who studies the C++ memory model, likes to give this example of undefined behaviour: "You could launch the missiles.")

Generating an error in such cases sounds like the most organized (precise) option. Writing the rules for this, however, would get rather tedious. More later.

6.2.1 Examples of undefined, unspecified, and implementation-dependent behaviour

- **OCaml**: Feels like one compiler (you download "ocaml", not two different compilers) but has two "back ends": one that generates machine code, and one that generates OCaml virtual machine bytecode. One back end evaluates function application left to right; the other evaluates function application right to left. If you care about order of evaluation, you need to use OCaml's `let`.
- **C**: Arithmetic overflow is undefined for signed integers. For unsigned integers, it must "wrap around". This seems to be because C predates the consensus that computer architectures should use "two's complement" to represent integers.[1]
- **C** (and many other languages): The size of an `int` was completely unspecified in 1970s/1980s C, and is now partly specified. C99 says that an `int` must be at least 16 bits—well, not quite. Rather, it

[1] stackoverflow.com/questions/18195715/why-is-unsigned-integer-overflow-defined-behavior-but-signed-integer-overflow-is

must be able to represent values between -32767 and 32767. In two's complement representation, 16 bits also gives you -32768.

- **C++**: On parallel architectures, which is most of them now that most CPUs have multiple cores, the C++ "memory model"—that is, the guarantees C++ offers about when code running on one core can actually see the effects of code on other cores—is... interesting.

If my memory serves (and if this hasn't changed in the last, oh, 20 years), Java has an unusual and refreshing shortage of undefined behaviour, which was probably motivated by the goal of "mobile code": a Java program should run anywhere with the same behaviour.

Exercise 4. Do some digging (a few Google searches may be enough) and read about undefined behaviour in your favourite (or least favourite) language. If you find something interesting, surprising, or horrifying, and you probably will, post a note on Piazza.

Exercise 5. (Not an exercise you can expect to actually *do*; just something to think about.) Suppose your programming language allows you to spawn threads that communicate with each other. How would you write an evaluation semantics for such a language?