CPSC 311: Definition of Programming Languages: Taxonomy of languages

"Oh, Captain! is there *no hope left?*"

("`11-taxonomy`")

Joshua Dunfield
University of British Columbia

October 6, 2015

Topics discussed

- categorizing syntax: "formal languages"; "C-like"; Algol-60
- categorizing semantics
- orthogonal language features

1 Categorizing languages: syntax

A language consists of syntax and semantics. Semantics consists of dynamic semantics (perhaps defined using a big-step semantics $e \Downarrow v$ or a small-step semantics $e1 \longrightarrow e2$) and static semantics. We'll jump into static semantics later this week.

I'm not very interested in syntax. However, syntax can be classified (somewhat) usefully using the theory of *formal languages*, in which "language" refers only to syntax: a "language" in that sense is simply a set of strings that are considered syntactically valid. Languages, or rather syntaxes, can be organized into the Chomsky hierarchy, first with *regular languages*, then *context-free languages*, then *context-sensitive languages*, and finally unrestricted languages. Each level in the hierarchy has a corresponding kind of automaton that *recognizes* that language, that is, the automaton determines whether the string is in the language (is syntactically valid). (Finite automata can recognize regular languages, pushdown automata can recognize context-free languages. . .) This is of relatively little interest to me, partly because even large programming languages have context-free syntax.

"BNF" is a specific notation for writing a context-free grammar.

(This general rule that PLs have context-free syntax has some exceptions. Parsing Perl is not context-free, because it's undecidable: `http://www.perlmonks.org/?node_id=663393`.)

Less formally, categories such as "C-like syntax" are easily recognized by humans, but carry little information: C, Java, and JavaScript all have C-like syntax, but their *semantics* are extremely different.

(Aside: Algol-60 had the idea, now largely forgotten, of explicitly distinguishing the notation used to (informally) *define* the language from the notation that programmers would type in. This was partly because there was no broadly-recognized standard character encoding—ASCII wasn't defined until 1963—but it meant that Algol-60 didn't try to forbid programmers from using certain "reserved words" or "keywords", because the language's creators assumed that each Algol compiler would define its own mapping from the "local" notation to the Algol notation used in the report. (The actual reason they adopted this idea: a violent disagreement about decimals; see Wexelblat 1981, *History of Programming Languages (I)*, p. 126). If this idea hadn't been forgotten, we might now be using editors and IDEs that could automatically switch between keywords in English and keywords in other languages. And perhaps between decimal separators.)

2 Categorizing languages: semantics

If categorizing syntax isn't that interesting, can we categorize semantics?

Yes, but maybe not with the usual categorization. Let's try the usual one anyway.

Imperative	**Object-oriented**	**Functional**	**Logic**
Fortran	Simula	Racket	Prolog

was partly because there was no broadly-recognized standard character encoding—ASCII wasn't defined until 1963—but it meant that Algol-60 didn't try to forbid programmers from using certain "reserved words" or "keywords", because the language's creators assumed that each Algol compiler would define its own mapping from the "local" notation to the Algol notation used in the report. (The actual reason they adopted this idea: a violent disagreement about decimals; see Wexelblat 1981, *History of Programming Languages (I)*, p. 126). If this idea hadn't been forgotten, we might now be using editors and IDEs that could automatically switch between keywords in English and keywords in other languages. And perhaps between decimal separators.)

Categorizing languages: semantics

If categorizing syntax isn't that interesting, can we categorize semantics?

Yes, but maybe not with the usual categorization. Let's try the usual one anyway.

imperative | not using state

Imperative procedural state	Object-oriented	Functional	Logic
		Scheme ?Lisp	
Fortran	Simula-67	Racket	Prolog
C ?Java	Java	Haskell Mercury =	Mercury
?COBOL	JavaScript	? Perl	Twelf
assembly	C++	? JavaScript	
Algol-60	Python	? Python	
Algol-68	C#	Java8 C++11	
Pascal	Smalltalk	? C#	
Modula-2	Objective-C	F#	
GLSL	Ruby	Ruby	
? Shell	Swift	Standard ML	
BASIC	Scala	OCaml	
C++	OCaml	Scala	
	Self	Smalltalk	
		~~WAE~~	
		~~AE~~	
		Fun	
		Fun++	

3 Categories: fuzzy at best

Essentially no languages fit neatly into these categories.

Explicitly "hybrid" languages, like OCaml (functional + OO), Mercury (functional + logic), should be expected to fall into more than one category, but really, no language fits neatly into these categories.

Everyone agrees that Racket is functional, but it has mutable data—but mostly not by default. Same for Standard ML and OCaml. In Lisp, mutable data is (was?) default, but Lisp otherwise "feels" functional?

Haskell doesn't have mutable data. . . but a lot of machinery, idioms, and libraries have been developed (and are extensively used) to let Haskell programmers pretend that it does!

OCaml has objects, but you don't have to use them and lots of OCaml programmers never do

C and C++ have everything mutable by default, but you can write `const`.

Java has "base types", like `int`, that aren't object-oriented at all.

Some OO languages (Self, Go?, . . . ?) don't have classes.

So these categories are really about default behaviours, and (even more) about what programmers actually do most of the time:

- C programmers tend to use mutation, even when they don't have to.
- Racket, SML, OCaml programmers tend to avoid mutable state "by default".
- OCaml programmers tend to avoid using objects.
- Java programmers (I assume?) tend to use objects even when they could use base types.
- Curiously, Haskell programmers seem pretty fond of the machinery developed to let them pretend that Haskell has mutable state. . .

4 Categorizing particular language features

Accepting that the categories listed above only suggest a kind of probability distribution on whether a particular feature is present, or how a particular feature works, we can instead ask more tractable specific questions, like "what evaluation strategy does language X use?"

But we should be aware that we're probably asking "what is the *default* evaluation strategy in language X?" For example, Scala lets you use the expression strategy, but only if you explicitly ask. Racket and SML use the value strategy, but it's not hard to simulate the expression strategy, just slightly annoying. (Live-code this?) Haskell uses lazy evaluation (related to the expression strategy), but you can get the value strategy by explicitly asking for it.

With that caveat, we can ask specific questions about "real" mainstream (and, usually, imprecisely defined) languages, and learn something (even if it's not truly precise) by comparing their features to the "equivalent" features in a small, idealized language like Fun++.

(I *would* be very comfortable putting Fun and Fun++ under the "Functional" heading.)

4.1 Categorizing Fun's variables

In addition to the value strategy (commonly known as "call by value") and expression strategy (commonly known as "call by name") for functions, we can consider whether Fun[++] has mutable state. Here, we can comfortably say it doesn't, because we can (and have) defined the dynamic semantics of Fun using substitution. When we substitute for the identifier bound by a lam, that identifier disappears, leaving no trace of its existence. In the body after substitution, we can't tell which expressions resulted from substituting for x and which expressions didn't. When we apply

$$(\mathsf{lam}\ x\ (\mathsf{add}\ (\mathsf{id}\ x)\ (\mathsf{num}\ 1)))$$

to (num 1), substitution gives

$$(\mathsf{add}\ (\mathsf{num}\ 1)\ (\mathsf{num}\ 1))$$

which has no sign of which (num 1) was originally (id x).

So we can conclude that Fun's variables (identifiers) are *immutable*, and that adding mutable variables would require us to change the semantics. (We will almost certainly do this later in 311.)

4.2 Categorizing Fun's functions

Functions are classified according to how "first-class" they are—essentially, whether they are values that can be passed around and used to construct other values (like integers can be passed around and used to construct other integers, or binary trees can be passed around and used to construct other binary trees).

("First-class" is standard terminology, but misleading, because it suggests that a "first-class function" is somehow special, when it's really the opposite: a first-class function is *a completely ordi-*

nary value. It's languages without first-class functions that have separated their functions from the rabble of ordinary values.)

Here, we again have a clear answer: functions in Fun are values, with no restrictions: they can be passed as arguments to other functions, and returned as results. As we add features to Fun, we should verify that they've kept this status. For example, the pairs in Fun++ don't affect this status, because pairs can hold any values, including functions.

When two language features do not interfere with or affect each other, we can say they are *orthogonal*, by a kind of geometric analogy. This is a vague definition, but I don't know of a better one. More precisely, we can say that two features are *defined orthogonally* if their definitions are all independent. (Warning: I *think* this is what other people mean by "defined orthogonally", but I'm not completely sure. I'm more sure that this is a useful definition and that it's the one we'll use in 311.)

Independence is not an entirely precise notion. Some cases are pretty clear: in Fun, functions and numbers are defined independently, because none of the rules for numbers and arithmetic mention any of the abstract syntax for functions, and none of the rules for `lam` and `app` mention any of the abstract syntax for numbers.

Some cases are less clear. I *want* to say that in Fun, functions and `with` are defined independently, because—again—the rules for functions don't mention `with`, and the rule(s) for `with` don't mention `lam` and `app`. But you could argue that both functions and `with` depend on the definition of *subst*. . . which mentions *all* the variants of the abstract syntax.

■ **Exercise 1.** Which new features in Fun++ (Assignment 2) are independent of which other features? (Because independence is not really precise, no answer can be perfectly right or perfectly wrong.)

Earlier, I almost said "when two language features do not interact with each other", but that could be misleading. Fun++ allows you to mix functions and pairs as much as you like: a function can take a pair as an argument, the pair can contain functions, and you can return a pair of functions. This demonstrates a key benefit of orthogonal designs: the combination of features, though defined separately, leads to a language that "subsumes" (maybe with a little added sugar) other, non-orthogonal features. If you can pass pairs as arguments, you are very close to allowing functions that take multiple arguments. But you got there by starting with the simplest possible form of function (a single argument to a single result), rather than saying, "well, we'll make functions take different numbers of arguments, so we have to think about whether a given function is being called with the right number of arguments. . . ".

■ **Exercise 2.** Given Fun plus pairs, how would you add multiple-argument functions as syntactic sugar?

■ **Exercise 3.** Given Fun without pairs, could you add multiple-argument functions as syntactic sugar? If so, how?