2 Formal Descriptions

Of programming language syntax and semantics

Mark Armstrong, PhD Candidate, McMaster University

Fall, 2019

1 Preamble

1.1 Notable references

- Concepts of Programming Languages, Sebesta, 10th ed.
  - Chapter 3 — Describing Syntax and Semantics
- Programming Languages and Operational Semantics: A Concise Overview, Fernández
  - Section 1.3.3 — Syntax
  - Section 1.3.4 — Semantics
- Concepts, Techniques, and Models of Computer Programming, Van Roy & Haridi
  - Section 2.1 — Defining practical programming languages

1.2 Update history

Sept. 16 • Original version posted

1.3 Table of contents

2 Expressions and statements; side effects and impurity

Before we can discuss syntax and semantics, we must differentiate the notions of

- an expression and
- a statement.

Recall that, in mathematics

- the term "expression" refers to a (syntactically correct, finite) combination of symbols.
  - Other terms used for this include "term".

Whereas

- the term "statement" refers to a specific kind of expression which denotes a truth value.
  - Other terms used for this include "formula".

2.1 Our definition of expression and statement

In the discussion of programming languages,

- the term "expression" refers to a (syntactically correct, finite) combination of lexemes (symbols), and
- the term "statement" refers to a specific kind of expression which has a *side effect*.
  - I.e., we understand the term "statement" as "imperative statement".
  - Other terms for this include "command".

  Usually we specifically use the term
- *expression* to mean an expression for which we are interested in its value, and specifically say
- *statement* when we are more interested in the expression's effect.

Sometimes we are interested in both the value and the effect.

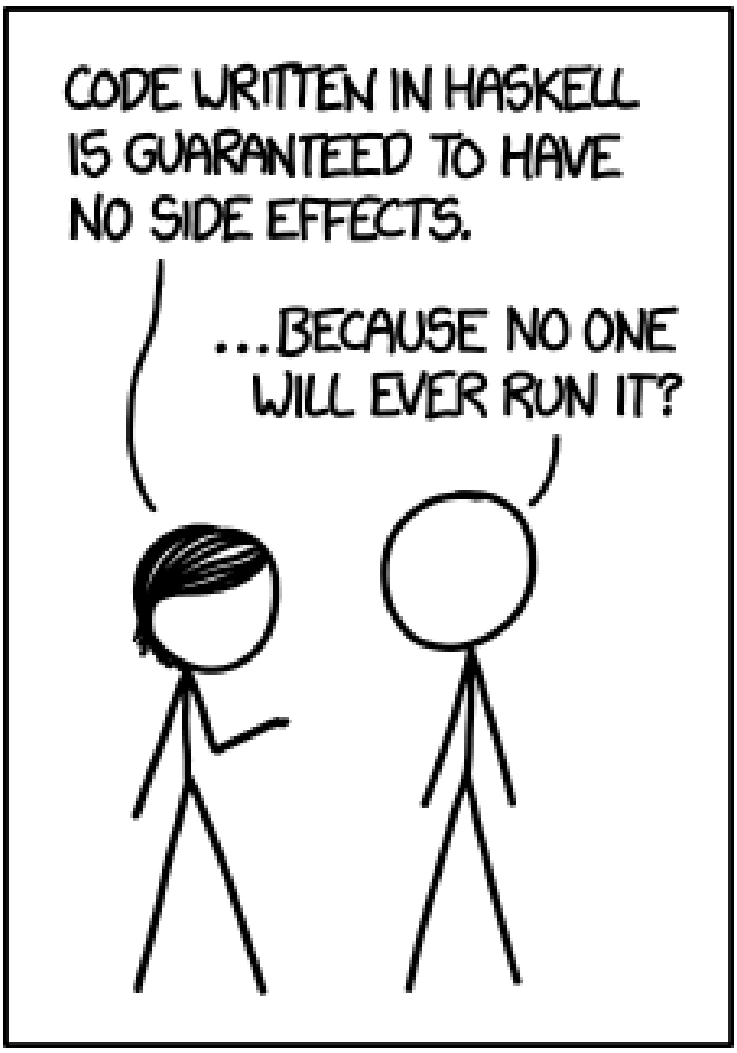
2.2 Side effects

A *side effect* of a program is any change to non-local state.

- Changes to the program's memory space are not side effects (of the program).
  - We can also talk about side effects of *parts* of programs.
    - In this case changes to the program's memory space may be side effects, if the change is to variables not local to the part under consideration.

  Examples of side effects:
- Modifying files.
- Writing to standard output or writing to the display.
- Sending messages over a network.

2.3 "Code written in Haskell is guaranteed to have no side effects"



> The problem with Haskell is that it's a language built on lazy evaluation and nobody's actually called for it.
>
> — xkcd 1312 - Haskell

2.4 Purity

While we are discussing side effects, let us futher discuss *pure code*.

A unit of code is called *pure* if, in addition to having no side effects, the result of its evaluation is *always the same*.

We are usually interested in *pure functions* in particular.

- A pure function's output depends only on its input.
- In mathematics, all functions are pure.
- If a function's output depends upon other factors, we could make those factors input to increase the function's pureness.

2.5 Pros and cons of side effects and impurity

Side effects and impurity may be considered a "necessary evil".

- (Interesting) programs *must* have some impurity and side effects.
  - Otherwise, they compute for no reason; we can never "see" any results.
- We cannot work entirely in a vacuum!
- Impurity and side effects make reasoning about code *much* more difficult, though!

A solution to this problem:

- Partition your code into
  - a *pure* back-end and
  - an *impure*, *side-effectful* front-end.

2.6 Don't take "side effect free" too literally

How "side effect free" can we be, exactly?

- Executing *any* code has an effect on the machine executing it.
  - The program counter, contents of main memory and the caches are changed, energy is expended for the processor, etc.
- But these inevitabilities are rarely our concern.
- We usually reason about an abstract machine, which hides such physical considerations from us.
  - How abstract the machine is varies throughout this course, based on the concept we're discussing.

3 Formal tools in the study of programming languages

We've said before, in order to facilitate running code on machines, programming languages must be *formal*.

There are two portions to this requirement.

- Describing the *syntax*; the form of expressions/statements.
- Describing the *semantics*; the meaning of expressions/statements.

Formal descriptions of semantics in particular are not always given!

- This decreases the reliability of the language.
  - It can never be clear if implementations (compilers, interpreters) correctly implement the language.
- Realistically, a "good enough" description may suffice.

3.1 Blurring the lines between syntax and semantics: static semantics

Unfortunately, we cannot cleanly divide elements of programming languages into the categories of syntax and semantics.

- The categorisation may depend upon the features of the language.
- *Typing* and *scope* may be considered syntactic or semantic.
  - Depending upon whether they are *static* or *dynamic*.
- Even if they are static, and could be considered syntax, such features are often called *static semantics*.

We will see an additional reason to segregate such features;

- static semantic rules cannot be expressed by the formal tools we use for syntax.
  - It's either impossible or prohibitively expensive.

4 Describing syntax

The standard tools for describing programming language syntax are *regular expressions* and *context-free grammars*.

Regular expressions may be used to describe *tokens*, (categories of) the smallest syntactic units of a language.

Content-free grammars are used to describe (most of) the form of programs.

- Static semantics cannot be described by CFGs.

Specifically, grammars in *extended Backus-Naur form* (EBNF) are usually used.

- A particular notation for grammars, extended with convenient features which do not increase expressivity (syntactic sugar).

As a brief example of EBNF (which we'll dissect soon),

```
digit = 1  2  3  4  5  6  7  8  9  0
nat = digit { digit }
int = [ - ] nat
exp = int  exp (+  *) exp
```

4.1 Tokens and lexemes

The smallest syntactic units of a programming language are called *lexemes*.

- Think of them as the *words* of the language.
  - E.g., `while`, `if`, `int`, `+`, `some-variable-name`, `a-function-name`, etc.

Categories of lexemes are called *tokens*.

- Comparing with natural languages, think of "prepositions", "pronouns", "conjunctions" etc.
- In programming, we have, e.g., *identifier*, *literal*, *operator*, *type name*.
  - Some categories have only a single member, without any additional information, e.g. *while*, *if*.

The first step in parsing a program is to convert it from plaintext to a list of tokens.

- *Tokenising.*
- At this stage, details are abstract; e.g., every identifier becomes just an identifier token (with its name attached in some way for later steps).
- Discards unnecessary information (whitespace, comments, etc.)

4.2 Tokenising

```
x = 0;
r = 1;
while (x < n) {
  r = r * x;
  x++;
}
```

```
id(x) eq lit(0) end_stmt
id(r) eq lit(1) end_stmt
while openbr id(x) op(<) id(n) closebr open_block
id(r) eq id(r) op(*)
id(x) end_stmt id(x) op(++) end_stmt
close_block
```

Disclaimer: this example is purely made up; it's not intended to be a completely accurate depiction of tokenising any particular language.

4.3 Extended Backus-Naur form

Consider again the example grammar using extended Backus-Naur form.

```
digit = 1  2  3  4  5  6  7  8  9  0
int = digit { digit }
exp = int  exp (+  *) exp
```

- Nonterminals are written in angle brackets `...`.
- The symbol `=` is used to begin a list of productions, rather than → or .
- Braces `{ ... }` indicate their contents may be repeated 0 or more times.

  - We may write `{ ... }` to indicate *1 or more* repetitions.

- Brackets `[ ... ]` indicate their contents are optional.
- The "mid" symbol may be used inside parentheses, `(...  ...)` to indicate choice in *part* of a production.

Notations differ!

- There is an ISO standard.
- We will not write a great number of grammars, so we will only use the notations introduced here.

4.4 Ambiguity

Recall that parsing a string (or deriving a string) using a grammar gives rise to a *parse tree* or *derivation tree*.

It is desirable to have a single parse tree for every program.

- We should not admit two syntactic interpretations for a program!

Three tools for removing ambiguity are

- requiring parentheses,
- introducing precedence rules, and
- introducing associativity rules.

4.5 Enforcing precedence and associativity with grammars

To enforce precedence using a grammar:

- Create a hierarchy of non-terminals.
- Higher-precedence operators are produced lower in the hierarchy.
- For instance,
  - An additive term can be a addition of multiplicative terms, which is an addition of literals, which can be the negation of a constant, variable or term.

To enforce associativity using a grammar:

- Left associative operators should be produced by left recursive non-terminals.
- And right associative operators by right recursive non-terminals.
- Operators of the same precedence must associate the same way!

4.6 Is addition associative?

Recall that addition is an associative operator.

- Meaning it is both left and right associative.

So the choice of whether addition in a language associates to the right or to the left may seem arbitrary.

- But numerical types in programming are not necessarily the same as numerical types in math!
- Addition of floating point numbers *is not associative*.
  - Consider a binary representation with two-digit coefficients.
  - $1.0 \times 2 + 1.0 \times 2 + 1.0 \times 2^2$ has a different value depending upon parenthesisation.

4.7 Abstract syntax

"Simple", ambiguous grammars do have a place in describing programming language syntax.

- Such grammars describe the *abstract syntax* of the language.
  - As opposed to *concrete syntax*.
- Consider programs as *trees* generated by the grammar for the abstract syntax of the language.
  - Trees do not admit ambiguity!
  - Such trees more efficiently represent programs.
    - The shape of the tree expresses structure.
    - Other unnecessary details may be left out.

4.8 Beyond context-free grammars: "static semantics"

For most interesting languages, context-free grammars are not quite sufficient to describe well-formed programs.

- They cannot express conditions such as "variables must be declared before use", and typing rules.
- It has been *proven* that CFGs are not sufficient.
  - At least some typing rules are possible to express, but prohibitively difficult.

Recall the Chomsky hierarchy of languages.

`Regular  Context-free  Context-sensitive  Recursive  Recursively enumberable`

- The properties we need could be described by *context-sensitive* grammars.
  - But they are unwieldy!
- Instead, use *attribute grammars*; a relatively small augmentation to CFGs.
  - Each non-terminal and terminal may have a collection of *attributes* (named values).
  - Each production may have a collection of rules defining the values of the attributes and a collection of predicates reasoning about those attributes.

4.9 An example attribute grammar

Consider this simple grammar.

```
S = A B C
A =   a A
B =   b B
C =   c C
```

Suppose we want to allow only strings of the form `abc`. There is no CFG that can produce exactly such strings. But we can enforce this condition using the above grammar augmented with attributes.

- Each of the non-terminals `A`, `B` and `C` are given an attribute `length`.

- To each production with `A`, `B` or `C` on the left side, we attach a rule to compute the `length`.
- The production `S = A B C` enforces the condition with a predicate.

```
S = A B C
Predicate: A.length = B.length = C.length

A =
Rule: A.length  0

A = a A
Rule: A.length  A.length + 1

B =
Rule: B.length  0

B = b B
Rule: B.length  B.length + 1

C =
Rule: C.length  0

C = c C
Rule: C.length  C.length + 1
```

In productions with multiple occurrences of the same non-terminal, we number the occurrences so we can easily refer to them in the rules/predicates.

5 Describing semantics

Unlike with syntax, there is not one universally used tool for describing programming language semantics.

In this course we will primarily consider *operational semantics*.

- A formal description of the meaning programs as a series of computation steps on an abstract machine.
  - The machine should be more abstract, and more easily understood, than assembly language.

  - But still "simpler" than the language.
  - Stack machines and state diagrams are good candidates.

Additional approaches include

- Denotational semantics.
  - The meaning of programs are *denoted* by mathematical objects.
    - Such as partial functions.
  - Have to consider *limits* and non-termination.
- Axiomatic semantics.
  - The meaning of a program is given by a precondition/postcondition calculus.
    - Such as `wp`; the "weakest-precondition" calculus.
  - Very useful for specification.

5.1 The kernel language approach

The "kernel language" approach to semantics can be used for languages with many features and constructs.

- Choose a small "kernel" set of features/constructs.
- Describe the remainder of the language in terms of that kernal language.
- The kernel language may be described using the formal approaches mentioned.
- *Concepts, Techniques, and Models of Computer Programming* takes this approach.

5.2 More to come...

We will return to the discussion of semantics later in the course.