The goal of semantics is to formalize the meaning of programming languages. Without semantics, programs are just pieces of syntax, that is, sequences of characters. What we need is to define what the program text means.

One way to find out about the meaning of constructs in a language is to read the language specification manual – that usually gives an informal description that explains the meaning of those constructs. The other alternative is to give a formal, mathematical definition of the language semantics. Advantages of doing so include the fact that formal descriptions are less ambiguous, more concise, and, more important, they allow us to write mathematical proofs for program properties that we're interested in. The drawback of deriving formal semantics is that they can lead to fairly complex mathematical models, especially if one attempts to describe all details in a full-featured modern language.

There are three main approaches to specify the semantics of programming languages:

- operational semantics: describes how a program would execute on an abstract machine;
- denotational semantics: models programs as mathematical functions;
- axiomatic semantics: describes program behavior using preconditions and postconditions;

There are different drawbacks between the three approaches, in terms of how much math they involve, how easy is it to use them in proofs, or to use them for an implementation.

1 A Simple Language: Expressions

To make it easier to understand semantics, we will start with a minimal programming language, that of arithmetic expressions. A program in this language is an expression. Executing a program means evaluating the expression to a number.

To describe the structure of this language we will use the following domains:

$$x, y, z \in \mathsf{Var}$$
$$n, m \in \mathsf{Int}$$
$$e \in \mathsf{Expr}$$

where Var is the set of program variables (i.e. strings of characters); Int is the set of constant integers; and Expr is the domain of expressions. The latter can be specified using a BNF (Backus Naur Form) grammar:

$$e ::= x \mid n \mid e1 + e2 \mid e1 * e2$$

This grammar specifies the syntax for the language. However, an immediate problem here is that this grammar is ambiguous. Take expression 1 + 2 * 3. One can build two abstract syntax trees:

There are several ways to deal with this problem. One is to rewrite the grammar for the same language to make it unambiguous. But that makes it more obscure. Another possibility is to extend the syntax with parentheses around expressions:

$$e ::= x \mid n \mid (e1 + e2) \mid (e1 * e2)$$

We can regard this grammar as the "concrete syntax" of the language, which specifies how to unambiguously parse a string into program phrases; as opposed to the original syntax, the "abstract syntax", which is ambiguous but describes the same language of expressions. In this course we will use the abstract syntax and assume that the abstract syntax tree is known (for details on parsing, grammars, and ambiguity elimination see the compiler course).

As a parenthesis, note that the syntactic structure of expressions in this language can be compactly expressed in ML using datatypes:

```
datatype exp = Var of string  |  Int of int  |  Add of exp * exp  |  Mul of exp * exp
```

In a language like Java, expressing this structure would require a more complex declaration consisting of a class hierarchy:

```
abstract class Expr { }
class Var extends Expr { String name; ... }
class Int extends Expr { int val; ... }
class Add extends Expr { Expr left, right; ... }
class Mul extends Expr { Expr left, right; ... }
```

At this point we have defined the language syntax. We would like now to define the semantics, that is, the meaning of this syntax. In Operational Semantics we can do so by describing how a program would execute on an abstract machine. Such an execution would consists of successive reductions of an expression until we reach a number, which represents the result of the computation.

The state of the abstract machine is usually referred to as a configuration, and for our language it must include two pieces of information:

- a store (aka environment or state), which assigns integer values to variables, so that the execution of programs can look up these values.
- the expression left to evaluate.

Consider an example. Suppose we want to evaluate expression (x+2)*(y+1) in a store where x has value 4 and y has value 3. Then, the execution of the program can be seen as the following sequence of steps, where boxes represent configurations (i.e., states of the abstract machine):

$$\boxed{\begin{array}{c} x=4, y=3 \\ \hline (y+2)*(x+1) \end{array}} \rightarrow \boxed{\begin{array}{c} x=4, y=3 \\ \hline (4+2)*(x+1) \end{array}} \rightarrow \boxed{\begin{array}{c} x=4, y=3 \\ \hline 6*(x+1) \end{array}} \rightarrow \boxed{\begin{array}{c} x=4, y=3 \\ \hline 6*(3+1) \end{array}} \rightarrow$$

$$\rightarrow \boxed{\begin{array}{c} x=4, y=3 \\ \hline 6*4 \end{array}} \rightarrow \boxed{\begin{array}{c} x=4, y=3 \\ \hline 24 \end{array}}$$

We want formalize this evaluation that consists of successive one-step reductions.