The structure of a language defines its syntax, but what defines semantics or meaning?

⇒ Behavior!

The most straightforward way to define semantics is to provide a simple interpreter for the programming language that highlights the behavior of the language,

⇒ Operational Semantics
Reading

- Sections 23.1 & 23.2 in MPL
Let’s develop an operational semantics for a simple programming language called ONE:

\[
\begin{align*}
\text{ONE:} & \quad <\text{exp}>^* ::= <\text{exp}> + <\text{mulexp}> \mid <\text{mulexp}> \\
& \quad <\text{mulexp}> ::= <\text{mulexp}> \ast <\text{rootexp}> \mid <\text{rootexp}> \\
& \quad <\text{rootexp}> ::= ( <\text{exp}> ) \mid <\text{constant}> \\
& \quad <\text{constant}> ::= \text{all valid integer constants}
\end{align*}
\]

Note: The grammar is unambiguous, both precedence and associativity rules of “standard” arithmetic are observed.

Do the following sentences belong to \(L(\text{ONE})\)? Why? Why not?
- \(s = 1 + 2 \ast 3\)
- \(s = (1 + 2) \ast 3\)
- \(s = a + 3\)
Abstract Syntax Trees

We want to define an operational semantics, i.e., an abstract interpreter for the language, but parse trees are not very convenient, too many non-terminal symbols ⇒ **Abstract Syntax Tree (AST)**

Transformation Rules:

- `<N>`
  - `T` ⇒ `T`
  - `A op B` ⇒ `op A B`

**Note:** This rule also applies to unary operators and operators with arity > 2.
Observation: The abstract syntax tree is a simplified form of the parse tree: same order as the parse tree, but no non-terminals.
What happens to parentheses in the AST representation of a program?

They are not needed!

ASTs naturally represent associativity and precedence relations.

Consider: \((1 + 2) \times 3\)

Parentheses do not contribute to computations, therefore the following tree transformations can be applied:
We can represent ASTs in Prolog:

```
+  \Rightarrow  plus(A,B)
```

```
*  \Rightarrow  times(A,B)
```

```
c  (constant)  \Rightarrow  const(c)
```

```
+  1  *  2  3  \downarrow  plus(const(1),times(const(2),const(3)))
```
ONE: Prolog Interpreter

A simple interpreter that computes a semantic value for syntactic constructs, the computation of this semantic value can be interpreted as the behavior: myeval / 2, AST input and semantic value as output.

```prolog
myeval(plus(X,Y),Value) :-
    myeval(X,XValue),
    myeval(Y,YValue),
    Value is XValue + YValue.

myeval(times(X,Y),Value) :-
    myeval(X,XValue),
    myeval(Y,YValue),
    Value is XValue * YValue.

myeval(const(X),Value) :- Value = X.
```

?- myeval(const(1),X).
X = 1
Yes

?- myeval(plus(const(1),const(2)),X).
X = 3
Yes

?- myeval(plus(const(1),times(const(2),const(3))),X).
X = 7
Yes
With our semantics (interpreter) in place we can now compute the semantic value of any expression program, e.g.

\[
\text{myeval(plus(const(1),times(const(2),const(3))),X).}
\]

\[
X = 7
\]
Exercises

- Extend the grammar for language ONE with the subtraction operator
- Extend the operational semantics appropriately, e.g.,
  - \( 6 - 3 \) should give the value 3
  Assume that the abstract syntax of this operator is \( \text{sub}(x, y) \).
- Compute the semantic value for the following expressions:
  - \( \text{sub}(3, 1) \)
  - \( \text{sub}(4, 2) \)
Exercises

- Extend the grammar for language ONE with the ‘!’ factorial operator.
- Extend the operational semantics appropriately, e.g.,
  - 3! should give the value 6
  Assume that the abstract syntax of this operator is fact(x).
- Compute the semantic value for the following expressions:
  - fact(3)
  - fact(4)