Formal Semantics

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

$\Rightarrow$ Behavior!

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

$\Rightarrow$ Operational Semantics

Operational Semantics

Let’s develop an operational semantics for a simple programming language called *ONE*;

ONE: <exp>* ::= <exp> **+** <mulexp> | <mulexp>
<mulexp> ::= <mulexp> ***** <rootexp> | <rootexp>
<rootexp> ::= **(**<exp>**)** | <constant>
<constant> ::= all valid integer constants

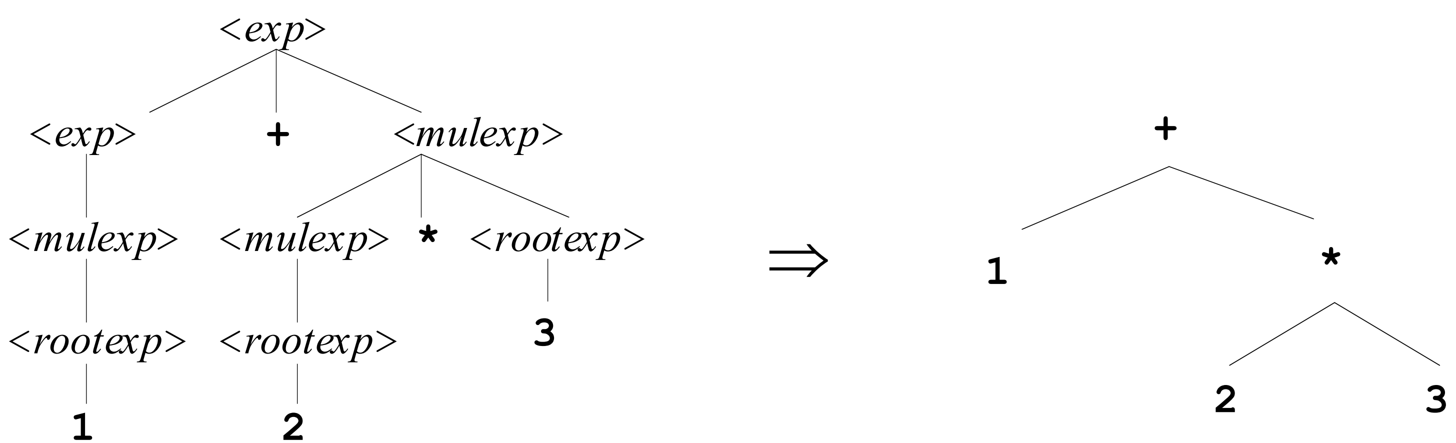
Note: The grammar is unambiguous, both precedence and associativity rules of “standard” arithmetic are observed.

Do the following sentences belong to *L*(*ONE*)? Why? Why not?

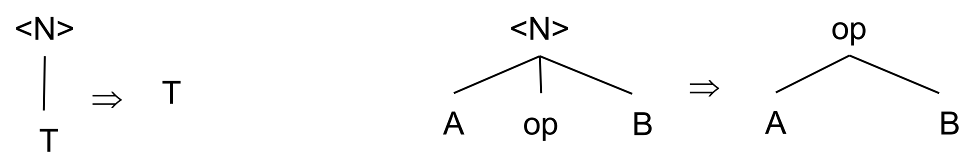
s = 1 + 2 * 3
s = (1 + 2) * 3
s = a + 3

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 $\Rightarrow$ <u>Abstract Syntax Tree</u> (AST)



<u>Transformation Rules:</u>



Note: This rule also applies to unary operators and operators with arity > 2.

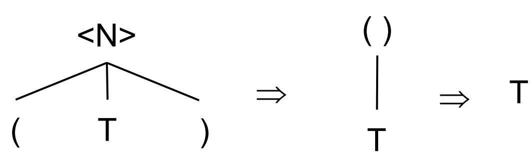
Observations

Definition: An abstract syntax tree is a finite, labeled, directed tree, where the internal nodes are labeled by operators, and the leaf nodes represent the operands of the node operators. -Wikipedia, 2006

Observation: The abstract syntax tree is a simplified form of the parse tree: same order as the parse tree, but no non-terminals.

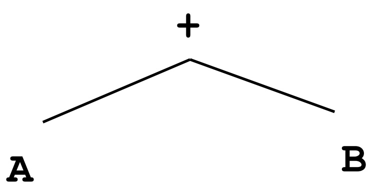
ASTs & Parentheses

- What happens to parentheses in the AST representation of a program?
- They are <u>not needed</u>!
- ASTs naturally represent associativity and precedence relations.
- Consider: (1 + 2) * 3
- Parentheses do not contribute to computations, therefore the following tree transformations can be applied:

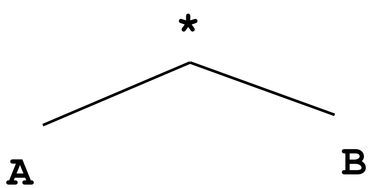


Prolog ASTs

We can represent ASTs in Prolog:

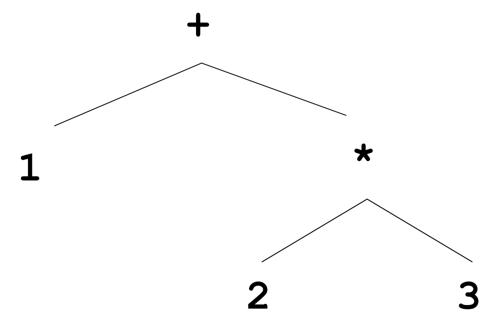


⇒ plus(A,B)



⇒ times(A,B)

`c` (constant) ⇒ const(c)



⇓

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.

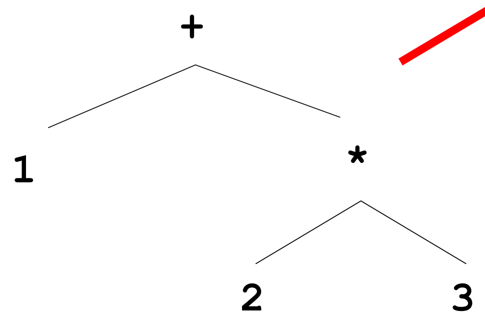
```
myeval(plus(X,Y),Value) :-
        myeval(X,XValue),
        myeval(Y,YValue),
        Value is XValue + YValue.

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
```

Semantics of Expressions

- With our semantics (interpreter) in place we can now compute the semantic value of any expression program, e.g.

1+2*3



plus(const(1),times(const(2),const(3)))

```
?- 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 sub(x,y).
- Compute the semantic value for the following expressions:
  - sub(3,1)
  - 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)