

## 10.1.1 A simple expression grammar

In this section, we extend the grammar example with code to represent a complete grammar, a parser, and an evaluator. The reader may skip this section and the subsequent ones on the expression parser and the abstract syntax tree during the first reading of this book.

We use a grammar for describing arithmetic expressions of digits using '+', '\*' and parentheses:

```
Start: <Exp>
<Exp> ::= <Exp> "+" <Term> | <Term>
<Term> ::= <Term> "*" <Primary> | <Primary>
<Primary> ::= <Number> | "(" <Exp> ")"
<Number> ::= <Number> <Digit> | <Digit>
<Digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

The start symbol of the grammar is the nonterminal <Exp>.

The nonterminal symbols are: <Exp>, <Term>, <Primary>, <Number>, and <Digit>.

The terminal symbols are: "+", "\*", "(", ")" and the digits "0" - "9".

We declare an ExpressionGrammar as a sub of Grammar:

```
ExpressionGrammar: obj Grammar
  -- declaration of symbols
  -- declaration of rules
  -- specification of start symbol
```

The symbols are declared as follows:

```
expSy: obj Nonterminal("Exp")
termSy: obj Nonterminal("Term")
primarySy: obj Nonterminal("Primary")
numberSy: obj Nonterminal("Number")
digitSy: obj Nonterminal("Digit")
add: obj Terminal("+")
mult: obj Terminal("*")
leftB: obj Terminal("(")
rightB: obj Terminal(")")
```

To generate the rules, we introduce a method for each nonterminal. The one for <Exp> looks as follows:

```
mkExpRule: addRule(expSy)
  S: ref SymbolList
  S := SymbolList.insertList((ExpSy,add,termSy))
  R.alternatives.insert(Alternative(S))
  S := SymbolList.insertList(termSy)
  alt := Alternative(S)
  R.alternatives.insert(alt)
```

As can be seen, `mkExpRule` is a submethod of the method `addRule`, which we have added to class `Grammar`. In addition, we have added class `SymbolList` with a `print` method:

```
class Grammar:
  ...
  class SymbolList: OrderedList(#Symbol)
    print:
      scan
        current.print
  addRule(L: ref Nonterminal):
    R: ref Rule
    R := Rule
    R.leftSide := L
    inner(addRule)
    rules.insert(R)
```

The method `addRule` has the leftside of the rule to be added as a parameter `L`. It generates a `Rule`-object and assign its reference to `R`, and then assigns `L` to the `leftSide` of `R`. Then it executes `inner(addRule)` implying that the mainpart of `mkExpRule` is executed. When returning from `inner`, the `Rule` `R` is inserted into the list of rules.

The expression `SymbolList.insertList((ExpSy, add, termSy))` may need an explanation:

1. First `SymbolList` is evaluated creating a `SymbolList`-object returning a reference to this newly created object.
2. Then `insertList((ExpSy, add, termSy))` is invoked on the reference to the new `SymbolList`-object.
3. The parameter of `insertList` is an array and the argument of the invocation is the array-literal `[(ExpSy, add, termSy)]` - array-literal is introduced in section . `insertList` inserts each element of the array in the newly generated `SymbolList`-object.

To complete the generation of `ExpressionGrammar`, we may add methods similar to `mkExpRule` for the other nonterminals, but leaves this an exercise for the reader. The `ExpressionGrammar` then looks as follows:

```
ExpressionGrammar: obj Grammar
  -- declaration of symbols
  ExpSy: obj Nonterminal("Exp")
  ...
  -- declaration of rules
  mkExpRule: addRule(ExpSy)
  ...
  mkTermRule: addRule(TermSy)
  ...
  mkPrimaryRule: addRule(PrimarySy)
  ...
  mkNumberRule: addRule(NumberSy)
  ...
  mkDigitRule: addRule(DigitSy)
  ...
```

```
-- specification of start symbol
start := ExpSy
mkExpRule
...
```

## **Parser and abstract syntax tree**

Next we show how to write a parser for the expressions of our grammar and how to represent an expression by means of an abstract syntax tree. The reader may skip these sections during a first reading.