## Der Parser

Der Parser ist der Teil des Programms, der einen Eingabestring in eine Form überträgt, die vom Computer leicht verarbeitbar ist, den Syntaxbaum (Beispiel s. Fig. 1).
Dies findet in der Klasse `Parser` statt.

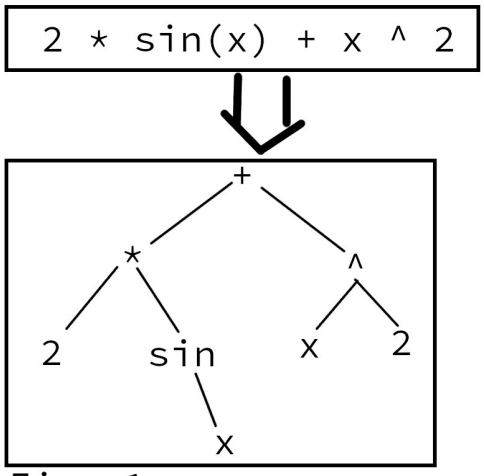

Fig. 1

Dieser Vorgang lässt sich wiederum in zwei Teilschritte unterteilen, die lexikalische und die syntaktische Analyse.
Bei der lexikalischen Analyse wird die Zeichenkette zunächst in seine einzelnen relevanten Bestandteile zerlegt, die Tokens. So würde beispielsweise der String aus Fig. 1 zu einem Array verarbeitet werden, dass in etwa so aussieht:

```
{ number: 2, *, func_id: sin, (, x, ), +, ^, number: 2 }
```

Diese Funktion wird im Quellcode durch die Methode `Parser.lex()` implementiert.

Das Array wird anschließend in der syntaktischen Analyse weiterverwertet. Bei der syntaktischen Analyse wird aus der Folge von Tokens der Syntaxbaum zusammengesetzt, aber auch auf Syntaxfehler überprüft. Dabei wird eine Reihe an Produktionsregeln befolgt, die Grammatik (siehe nächste Seite). Der Syntaxbaum ist ein einfacher Baum der aus Knoten mit 0 bis 2 Unterbäumen besteht. Die abstrakte Überklasse der Knoten ist `SyntaxNode`, von der alle konkreten Syntaxknoten wie `AddNode`, `SubNode`, `NumberNode` etc. abgeleitet sind.

Die von diesem Parser verwendete Grammatik in EBNF lautet wie folgt:

```
<expr>  ::=  <term> {( "+" | "-" ) <term> }
<term>  ::=  <pow>  {( "*" | "/" ) <pow>  }
<pow>   ::=  "-" <pow>
          |  <fact> "^" <pow>
          |  <fact>
<fact>  ::=  number
          |  func_id "(" <expr> ")"
          |  "(" <expr> ")"
          |  x
```

Durch diese Grammatik ist eine formale Sprache definiert, die beliebige mathematische Ausdrücke beinhaltet. Hier handelt es sich auf Grund der etwas spezielleren Produktionsregeln `<expr>` und `<term>` streng genommen nicht mehr um eine simple LL(1)-Grammatik, jedoch kann sie trotzdem mithilfe eines kleinen Tricks (Verwendung einer while-Schleife, siehe Quellcode `Parser.parseExpr()` und `Parser.parseTerm()`) durch einen recht einfachen Recursive-Descent-Parser implementiert werden. Jede der Produktionsregeln ist im Quellcode durch eine eigene Methode implementiert, die sich gegenseitig rekursiv aufrufen können.
Endet während der syntaktischen Analyse der Engabestring unerwartet, oder tritt ein unerwartetes Token auf, also folgt beispielsweise auf einen Operator “*” ein weiter Operator “/”, was quasi ein Zustandsübergang wäre der durch die Grammatik nicht gedeckt ist, so wird eine `SyntaxException` geworfen.
Dieses Objekt beinhaltet den Start- und Endindex sowie Substring des fehlerverursachenden Symbols im Eingabestring.

Diese Implementationsdetails können von dritten aber auch schlicht außer Acht gelassen werden, da der Parser über ein simples Interface verfügt und somit als Blackbox behandelt werden kann. Es muss nur ein Objekt der Klasse `Parser` instanziert werden, anschließend lässt sich der ganze Rest einfach über den Aufruf der Methode `buildSyntaxTree(String string)` geregelt werden, welche dann entweder die Wurzel des Syntaxbaumes (Typ `SyntaxNode`) zurückgibt, oder eine `SyntaxException` mit entsprechenden Details wirft.

## Syntaxbaum

Hat man nun erst einmal den Syntaxbaum zusammengesetzt, ist das Auswerten relativ trivial. Die Klasse SyntaxNode gibt eine abstrakte Methode `eval(double x)` vor, welche von den abgeleiteten Klassen entsprechend implementiert wird. Ruft man nun `eval()` auf, wird der Baum rekursiv durchschritten und ausgewertet. Der Parameter `double x` ist dabei der Wert der Variablen x. Der Rückgabewert ist dann der zugehörige Funktionswert.

Zusätzlich verfügt jeder `SyntaxNode` über die Methode `print()`. Diese ermöglicht es, den Syntaxbaum in der Konsole auszugeben, um ihn auf Korrektheit zu überprüfen.