Dummie Mitglied 16:25:33 06.06.2010 Titel: **Interpreterbau - Ein Anfang**

Inhalt

1 Einleitung
Die Entwicklung eines Compilers (Compilerbau oder auch Übersetzerbau) ist schon lange keine Magie mehr: Tatsächlich handelt sich um ein sehr interessantes Themengebiet in der Informatik. Trotzdem besitzt es heutzutage noch immer den Ruf, dass es für einen "normalen" Menschen nicht möglich sei, selbst einen Compiler zu entwickeln.

Als interessierter Leser stellt man sich selbstverständlich die Frage: "Warum? Woher kommt das?".

Nun, früher war eben nicht alles besser. Computer hatten nicht so viel Leistung und der Speicher war natürlich auch stark begrenzt. Und doch hat man es geschafft, effiziente Programme zu schreiben. Dazu optimierte man an vielen Stellen, lagerte Programmteile aus und betrieb tatsächlich ein bisschen kreative "Zauberei". Diese Art Optimierung trat insbesondere bei Compilern auf, da an diese besondere Leistungsanforderungen gestellt wurden. Sie erfuhren also enorme Optimierungen, so dass die Lesbarkeit und Verständlichkeit des Quellcodes dadurch immer weiter abnahm.

Warf man dann als Neugieriger einen Blick auf einen solchen Quellcode, sah man sehr komplexen, unverständlichen Code.

Doch das muss heute nicht mehr sein!
Das alles liegt inzwischen in weiter Ferne und es hat sich viel getan. So wurden nicht nur die Vorgehensweisen weiterentwickelt, vereinfacht und verfeinert. Auch die Computer sind viel leistungsfähiger geworden und wir können getrost auf solche Optimierungen verzichten.

2 Motivation
Nach dieser Einleitung sollte klar geworden sein:
Auch Compiler sind letztendlich nur Computerprogramme. Wir werden uns in diesem Teil des Artikels auf die Entwicklung eines Matheparsers beschränken. Jedoch sollen damit die Grundsteine für mehr gelegt werden. Genauer gesagt soll in den weiteren Teilen dieser Artikelserie eine kleine Scriptsprache entstehen. Das dazu notwendige Wissen soll Schritt für Schritt aufgebaut werden. Dieses Wissen beschränkt sich nicht nur auf die eigentliche Programmierung, sondern soll auch zeigen wie man eine eigene Syntax definiert und später als Programm umsetzt.

Auf komplizierte theoretische Erklärungen wird möglichst verzichtet.
Stattdessen gib es viele funktionsfähige Beispielcodes, die die beschriebenen Konzepte näher bringen und implementieren - die Tür für eigene Experimente ist damit geöffnet.

3 Quellcodes
Ich habe mich bemüht, möglichst viel Quellcode direkt im Artikel zu zeigen. Doch natürlich kann ich nicht bei jeder kleinsten Änderung den gesamten Quellcode wiederholen. Das ist aber kein Problem, da der gesamte Quellcode als Download bereitsteht. Sollten also Unklarheiten entstehen, so kann ein Blick in den Quellcode nicht schaden.

Die in diesem Artikel verwendete Programmiersprache ist natürlich C++, aber es steht auch eine (fast) exakte Portierung in die Programmiersprache C# zur Verfügung.

Der Vorteil: Hier ist eine GUI gegeben, so dass die Funktionsweise besser visualisiert werden kann. Dabei wird sowohl das Ergebnis des Scanvorgans als auch das Ergebnis des Parsingvorgangs angezeigt. Hier kann also auch experimentiert werden.

4 Erklärung einiger Begriffe
Um den Zusammenhang der einzelnen Begriffe verständlicher zu machen, folgt eine kurze Erklärung:

EBNF
Mit der EBNF (Erweiterte Backus-Naur-Form) können Grammatiken definiert werden.
Bevor eine Grammatik für eine Sprache definiert wird, muss man sich aber überlegen, wie die Sprache überhaupt aussehen soll. Hat man dies geklärt, kann man beginnen die Grammatik als EBNF darzustellen. Dabei hilft diese Notation ungemein, da aus ihr sofort ersichtlich wird, welche Funktionen später programmiert werden müssen und aus welchen "Zeichen" die Sprache besteht.

Scanner
Nachdem die EBNF also aufgestellt wurde, kann ein Scanner entwickelt werden, welcher die Eingabe in kleine Teile "zerteilt". Man bezeichnet diese kleinen Teile als Tokens. Dabei wird aus der EBNF ersichtlich, welche Tokens der Scanner erkennen muss. Bei einem mathematischen Ausdruck sind das natürlich die Zahlen und Operatoren. Je nach Komplexität müssen auch noch weitere Tokens, wie z.B. Bezeichner (für Konstanten z.B. PI) aus der Eingabe extrahiert werden.

Parser
Der Parser steuert schließlich den Scanner an und erfragt immer das nächste Token.
Dabei wird überprüft, ob das erhaltene Token an dieser Stelle überhaupt gültig ist. Es findet also eine Überprüfung der Syntax statt. Bei Sprachen mit Variablen, etc. findet an dieser Stelle zudem die sog. semantische Analyse statt: Es wird überprüft, ob z.B. Variablen deklariert wurden.
Durch die Möglichkeit, Funktionen rekursiv aufzurufen, kann der Parser auch verschachtelte Ausdrücke ohne Mehraufwand parsen: Er ruft einfach die Anfangsfunktion erneut auf. Um weitere Optimierungen zu ermöglichen, wird beim Parsingvorgang häufig ein Abstrakter Syntaxbaum (siehe nachfolgende Erklärung) aufgebaut.

Abstrakter Syntaxbaum
Der Abstrakte Syntaxbaum ermöglicht es weitere Optimierungen durchzuführen. Auch die Codegenerierung kann so vom eigentlichen Parsing getrennt werden. Dabei stellt er das Ergebnis des Parsingvorgangs in einer baumartigen Struktur dar, die alle notwendigen Informationen beinhaltet.

Das C# Programm visualisiert diesen Vorgang für einen eingegebenen Ausdruck.

5 Erweiterte Backus-Naur-Form (EBNF)
Wie bereits erwähnt kann mit der EBNF eine Syntax definiert werde.

Dazu zunächst eine kleine Übersicht über einige (es gibt noch mehr) in der EBNF verwendeten Zeichen:

Code:

```
[b]Alternative[/b] |
[b]Optionale Wiederholung[/b] { }
[b]Gruppierung[/b] ( )
[b]Option[/b] [ ]
```

Eine Syntax kann dann beispielsweise so aussehen:

Code:

```
[b]Antwort[/b] = "ja" | "nein" ;
```

Das besagt, dass das Nichtterminalsymbol (Nichtterminalsymbole stehen links) Antwort entweder "ja" **oder** "nein" sein kann. Alle anderen Eingaben sind ungültig.

Man kann das ganze noch in je ein eigenes Nichtterminalsymbol "auslagern":

Code:

```
[b]Antwort[/b] = (AntwortPositiv) | (AntwortNegativ) ;
[b]AntwortPositiv = "ja"[/b] ;
[b]AntwortNegativ = "nein"[/b] ;
```

Will man nun weitere Möglichkeiten definieren, so geht das auch sehr einfach:

Code:

```
[b]Antwort[/b] = (AntwortPositiv) | (AntwortNegativ) ;
[b]AntwortPositiv[/b] = (AntwortPositivDeutsch) | (AntwortPositivEnglisch) ;
[b]AntwortPositivDeutsch[/b] = "ja" | "jo" ;
[b]AntwortPositivEnglisch[/b] = "yes" | "yeah" ;
[b]AntwortNegativ[/b] = (AntwortNegativDeutsch) | (AntwortNegativEnglisch) ;
[b]AntwortNegativDeutsch[/b] = "nein" | "ne";
[b]AntwortNegativEnglisch[/b] = "no" | "nope";
```

Man erkennt, dass man das ganze durch einsetzen oder "auslagern" umformen kann.
Auch Rekursionen sind möglich - wir machen davon später Gebrauch.

Soll eine Syntax definiert werden, die nur Zahlen zulässt und die keine 0 enthält, so definieren wir uns zunächst ein Symbol für alle Ziffern außer 0:

Code:

```
[b]AlleZiffernAußerNull[/b] = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

Und das Symbol an dem unsere Syntax startet, nennen wir sprechender Weise Start:

Code:

```
[b]Start[/b] = (AlleZiffernAußerNull) ;
```

Diese Syntax akzeptiert nur eine Ziffer, die nicht 0 ist. Geplant war aber eine Ziffernfolge, die keine 0 enthält.

Dazu modifizieren wir die Syntax folgendermaßen:

Code:

```
[b]Start[/b] = (AlleZiffernAußerNull) { (AlleZiffernAußerNull) } ;
```

Man könnte das so lesen: Das Symbol Start besteht aus einer Ziffer die nicht 0 ist. Gefolgt von keiner oder mehreren Ziffern, die nicht 0 sind.

Anders gesagt: Eine Zahl die keine 0 enthält.

Die Eingabe von 4453453 wäre also gültig, während die Eingabe 1234**0**41 ungültig ist, da sie eine 0 enthält.

Um dem Ziel, einen eigenen Matheparser zu entwickeln, näher zu kommen, definieren wir nun eine Syntax, die die Addition und Subtraktion von Zahlen erlaubt.

Zunächst definieren wir, was eine Zahl ist:

Code:

```
[b]Ziffer[/b] = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
[b]Zahl[/b] = (Ziffer) { (Ziffer) } ;
```

Eine Zahl ist also einer Ziffer, auf die optional weitere Ziffern folgen... logisch.

Jetzt definieren wir die eigentlichen Rechenoperatoren:

Code:

```
[b]Start[/b] = (Zahl) { ("+" | "-") (Zahl) } ;
```

Unsere Berechnung besteht aus einer Zahl, auf die optional ein + oder – und eine weitere Zahl folgt.
Die Eingabe von 500+12-300+12-1+5 wäre also gültig. Aber auch die Eingabe von 5 ist vollkommen korrekt.

Schließlich handelt es sich bei { ("+" | "-") (Zahl) } um eine **optionale** Wiederholung.

Natürlich möchten wir auch Multiplikationen unterstützen. Mit der richtigen Syntax ergeben sich Dinge wie Punktrechnung vor Strichrechnung automatisch.

Die gesamte Syntax sieht dann so aus:

Code:

```
[b]Start[/b] = [b](Multiplikation)[/b] { ("+" | "-") [b](Multiplikation)[/b] } ;

[b]Multiplikation[/b] = (Zahl) { [b]("*" | "/")[/b] (Zahl) } ;

[b]Zahl[/b] = (Ziffer) { (Ziffer) } ;

[b]Ziffer[/b] = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

Um das Konzept zu verstehen, betrachten wir nun den Verlauf bei der Eingabe 10.

Wir befinden uns zunächst beim Symbol Start. Anschließend begeben wir uns zum Symbol Multiplikation, danach zum Symbol Zahl und abschließend zum Symbol Ziffer. *Jedoch werden wir die Verarbeitung von mehreren Ziffern zu einer Zahl direkt in den Scanner implementieren. Das soll in der Praxis nicht die Aufgabe unseres Parsers sein.*

Wir "fallen" also bis zum Ende hindurch. Daraus lässt sich für uns eine simple Regel ableiten:
Umso weiter ein Symbol in der Syntax zurückliegt, umso eher wird es erreicht.

Das ist auch der Grund warum wir in Start die Addition/Subtraktion definieren. Dieses Symbol wird als letztes abgearbeitet. Denn vorher wird die Multiplikation abgearbeitet. So behandeln wir die Punkt- vor Strichrechnung korrekt.

Doch was wäre Mathematik ohne Klammern? Klammern haben die höchste Rangfolge, also müssen sie als erstes abgearbeitet werden. Sie landen damit in unserer Definition noch weiter hinten.

Wir müssen unsere Syntax erneut modifizieren. Das könnte so aussehen:

Code:

```
1 [b]Start[/b] = (Multiplikation) { ("+" | "-") (Multiplikation) } ;
2
3 [b]Multiplikation[/b] = [b](Klammer)[/b] { ("*" | "/") [b](Klammer)[/b] } ;
4
5 [b]Klammer[/b] = (Zahl) | [b]"(" (Start) ")" ;[/b]
6
7 [b]Zahl[/b] = (Ziffer) { (Ziffer) } ;
8
9 [b]Ziffer[/b] = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

Wenn man sich das genau ansieht, stellt man fest: Wir haben eine Rekursion mit dem Symbol Start in unserer Syntax. Diese Rekursion ist gewollt, da in einer Klammer auch z.B. wieder eine Klammer stehen kann. Damit wir auch mit Klammern und Zahlen umgehen können, vor denen ein Vorzeichen steht, müssen wir noch eine letzte kleine Änderung machen:

Code:

```
1 [b]Start[/b] = (Multiplikation) { ("+" | "-") (Multiplikation) } ;
2
3 [b]Multiplikation[/b] = (Klammer) { ("*" | "/") (Klammer) } ;
4
5 [b]Klammer[/b] = [b]["+" | "-"][/b] ((Zahl) | "(" (Start) ")") ;
6
7 [b]Zahl[/b] = (Ziffer) { (Ziffer) } ;
8
9 [b]Ziffer[/b] = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

Damit **kann** vor einer Klammer auch ein Vorzeichen stehen, aber es muss natürlich nicht.

Unsere Syntax ist an dieser Stelle fertig. Wir haben damit die Syntax für die Grundrechenarten, sowie Klammern aufgestellt.
Auch die Rechenregeln sind berücksichtigt. Das soll uns zunächst genügen, aber natürlich wollen wir auch ein Programm entwickeln, das nach dieser Syntax vorgeht und uns aus einer Eingabe ein mathematisch korrektes Ergebnis liefern kann.

Damit wir bei der Eingabe der Daten möglichst flexibel sind und z.B. Leerzeichen automatisch ignoriert werden, teilen wir unser Programm in zwei Programmteile (repräsentiert durch Klassen) auf:

- Scanner (auch Lexer genannt)
- Parser

*Zum vollständigen Verständnis ist es sehr wichtig, dass dieser Teil des Artikels (EBNF) gut verstanden wird. Eine gute Übung wäre es z.B. die Syntax für mathematische Terme selbst noch mal aufzustellen. Auch das manuelle anwenden auf einen mathematischen Term (angefangen von 5+5 bis 100 * 3 + (4 - 1 *)) mit Stift und Papier wäre für das Verständnis vielleicht förderlich.*

6 Scanner
Der Scanner "zerteilt" die Eingabe in sogenannte Tokens. Der Parser ruft diese Tokens nach und nach vom Scanner ab und verarbeitet sie nach der von uns aufgestellten Syntax. Dabei bereinigt der Scanner die Eingabe auch von allen unerwünschten Zeichen, zum Beispiel Leerzeichen, Tabulatoren und auch Zeilenumbrüchen.

Wir beginnen nun mit der Programmierung des Scanners. Als erstes benötigen wir eine Enumeration mit den möglichen Typen von Tokens, damit wir diese unterscheiden können:

C++:

```
// Unsere Arten von Tokens
enum TokenType
{
    TT_PLUS, // +
    TT_MINUS, // -
    TT_MUL, // *
    TT_DIV, // /

    TT_LPAREN, // (
    TT_RPAREN, // )

    TT_NUMBER, // 12345

    TT_NIL // NIL (=Not In List) => Kein passendes Token gefunden
};
```

Damit wir später auch eine verständliche Fehlermeldung ausgeben können, legen wir noch ein Array an, das die Namen der Tokens als Zeichenfolge hält:

C++:

```
// Unsere Arten von Tokens als String (Für Fehlermeldungen)
static const char *TokenTypeStr[] =
{
    "TT_PLUS",
    "TT_MINUS",
    "TT_MUL",
    "TT_DIV",

    "TT_LPAREN",
    "TT_RPAREN",

    "TT_NUMBER",

    "TT_NIL"
```

```
15 };
```

Das Präfix TT_ steht hier für TokenType. Mit dieser Aufzählung können wir unsere Tokens unterscheiden. Dem TokenType TT_NIL kommt dabei eine ganz besondere Bedeutung zu: Sollte der Scanner das Ende der Eingabe erreicht haben oder kein Token erkennen, so bezeichnen wir dieses als TT_NIL.

Damit wir einem Token auch einen Wert zuweisen können (eine erkannte Zahlenfolge hat schließlich auch einen Zahlenwert) schreiben wir uns eine Klasse Token, in der wir neben dem TokenType auch den Wert speichern können:

C++:

```
 1 class Token
 2 {
 3    private:
 4       TokenType myType;
 5       int myValue;
 6
 7    public:
 8       Token(TokenType type = TT_NIL, int value = 0);
 9
10       TokenType getType() const { return myType; }
11       int getValue() const { return myValue; }
12
13       // Für Fehlermeldungen
14       const char *toString() const { return TokenTypeStr[myType]; }
15
16       // Zum Vergleichen
17       bool equal(TokenType type) const { return myType == type; }
18 };
```

Nehmen wir nun als Beispiel die Eingabe 10+3.

Es handelt sich dabei um folgende Tokens:

TT_NUMBER mit einem Wert von 10
TT_PLUS mit einem Wert von 0 (Standardwert. Schließlich braucht das Plus keinen Wert)
TT_NUMBER mit einem Wert von 3

Diese "Zerteilung" ist die Aufgabe unseres Scanners, der dazu eine Funktion getNextToken anbieten soll. Dabei wird die Eingabe Zeichenweise gelesen.
Wird zum Beispiel das Zeichen + gelesen, so wird das Token TT_PLUS angenommen und von der Funktion zurückgegben. Wird die Ziffer 1 gelesen, so wird so lange weitergelesen, bis keine Ziffer mehr folgt und die gesamte Zahl als Token TT_NUMBER zurückgegeben.

Auf diese Weise können wir unsere Eingabe für den Parser in Tokens verarbeiten, überflüssige Zeichen ignorieren und Fehlermeldungen ausgeben, falls unbekannte Zeichen auftreten.

Damit haben wir erst mal das Wichtigste zusammen und können uns nun an den Scanner wagen. Dabei handelt es sich auch um eine Klasse, die zunächst die Eingabe als std::string speichert. Außerdem weiß sie darüber Bescheid, an welcher Stelle wir uns in der Eingabe zurzeit befinden und welches Zeichen wir zuletzt gelesen haben.

Außerdem enthält die Klasse die angesprochene Funktionen getNextToken, sowie skipSpaces und readNextChar. Auf deren Implementierung und Sinn wird im Folgenden noch genauer eingegangen.

Doch zunächst die Klasse Scanner, welche - für viele vielleicht überraschend - erstaunlich klein ausfällt:

C++:

```
 1 class Scanner
 2 {
 3    private:
 4       std::string myInput;
 5       unsigned int myPos; // Aktuelle Position in der Eingabe
```

```
 6       char myCh; // Zuletzt gelesenes Zeichen
 7
 8   public:
 9       Scanner(const std::string& input);
10
11       Token getNextToken();
12
13   private:
14       void skipSpaces();
15       void readNextChar();
16 };
```

Als erstes betrachten wir die Funktion readNextChar. Wie der Name bereits vermuten lässt, liest diese Funktion das nächste Zeichen ein. Das ist zwar sehr simpel, aber man muss hierbei natürlich beachten, dass man nicht mehr liest, als die Eingabe lang ist:

C++:

```
 1 void Scanner::readNextChar()
 2 {
 3     // Am Ende der Eingabe angelangt?
 4     if (myPos > myInput.length())
 5     {
 6         myCh = 0;
 7         return;
 8     }
 9
10     myCh = myInput[myPos++];
11 }
```

Wenn das Ende der Eingabe erreicht wurde, wird das aktuelle Zeichen auf 0 gesetzt. Das ist für uns ein eindeutiges Merkmal, dass wir das Ende erreicht haben.

Eine weitere wichtige Funktion ist die Funktion skipSpaces. Diese Funktion sorgt dafür, dass Leerräume jeder Art (Leerzeichen, Tabulatoren, Zeilenumbrüche) ignoriert bzw. übersprungen werden:

C++:

```
void Scanner::skipSpaces()
{
    while (myCh == ' ' || myCh == '\t' || myCh == '\r' || myCh == '\n')
    {
        readNextChar();
    }
}
```

Damit der Scanner auch anfangen kann zu arbeiten, müssen im Konstruktor noch einige Variablen initialisiert und das erste Zeichen eingelesen werden:

C++:

```
Scanner::Scanner(const std::string& input) : myInput(input), myPos(0) // Initialisierung
{
    // Erstes Zeichen einlesen
    readNextChar();
}
```

Jetzt ist der Scanner bereit und es folgt das Herz unseres Scanners: Die Funktion getNextToken. Man kann den hier vorgestellten Scanner natürlich auf viele verschiedene Arten implementieren. Unter anderem wäre es denkbar, dass man zunächst **alle** Tokens einliest. Das kann wichtig sein, wenn man nachzusehen möchte/muss, was das nächstfolgende Token ist, ohne es tatsächlich zu lesen. Für uns ist das an dieser Stelle aber nicht relevant und wir lesen erst beim Aufruf der Funktion getNextToken die nächsten Zeichen ein und nicht vorher. Die andere Möglichkeit wird später aber auch noch vorgestellt.

Die konkrete Funktionsweise ist sehr einfach zu verstehen. Zunächst werden alle Zeichen übersprungen, die für uns unwichtig sind. Das übernimmt die Funktion skipSpaces. Anschließend betrachten wir das aktuelle Zeichen und entscheiden anhand von diesem Zeichen, wie damit umzugehen ist. Ist das Zeichen ein +, dann ist die Sache sofort klar: Es handelt sich um das Token TT_PLUS. Lesen wir eine Ziffer ein, dann müssen wir überprüfen, ob noch weitere Ziffern folgen und diese zu einer Zahl zusammenfassen. Der Einfachheit wird dafür ein Buffer vom Typ std::string verwendet. Die Ziffernfolge wird dann einfach mit atoi in eine Zahl konvertiert:

C++:

```
 1 Token Scanner::getNextToken()
 2 {
 3    std::string buf;
 4
 5    // Überflüssige Zeichen ignorieren
 6    skipSpaces();
 7
 8    switch (myCh)
 9    {
10       case '+':
11          readNextChar();
12          return Token(TT_PLUS);
13
14       case '-':
15          readNextChar();
16          return Token(TT_MINUS);
17
18       case '*':
19          readNextChar();
20          return Token(TT_MUL);
21
22       case '/':
23          readNextChar();
24          return Token(TT_DIV);
25
26       case '(':
27          readNextChar();
28          return Token(TT_LPAREN);
29
30       case ')':
31          readNextChar();
32          return Token(TT_RPAREN);
33
34       case '0': case '1': case '2': case '3': case '4':
35       case '5': case '6': case '7': case '8': case '9':
36          // Einlesen, solange es eine Ziffer ist
37          while (isdigit(myCh))
38          {
39             buf += myCh;
40             readNextChar();
41          }
42
43          // Der Einfachheit haben wir die Zahl zunächst in einen String eingelesen
44          // Dieser wird nun in ein Integer konvertiert
45          return Token(TT_NUMBER, atoi(buf.c_str()));
46
47       default:
48          if (myCh != 0)
49          {
50             std::cerr << "Error: nicht unterstuetztes Zeichen '" << myCh << "'" << std::endl;
51          }
52          break;
53    }
54
55    // Es wurde kein passendes Token gefunden. Mögliche Gründe dafür:
```

```
56    // - Eingabe enthält kein weiteres Zeichen mehr (Ende erreicht)
57    // - Eingabe besteht aus nicht unterstützten Zeichen
58    return Token(TT_NIL);
59 }
```

Hinweis: Einige Computerprogramme akzeptieren keine Zahlen mit führenden Nullen. Die Zahlenfolge 001 wäre also ungültig und nur die Zahlenfolge 1 korrekt. Unser Scanner akzeptiert jedoch auch Zahlen mit führenden Nullen!

An dieser Stelle ist der Scanner fertig. Er kann in einem Testlauf sehr einfach getestet werden:

C++:

```
 1 #include <iostream>
 2 #include <string>
 3 #include "Scanner.h"
 4
 5 int main()
 6 {
 7    std::string input = "10 + 5\n * \t10-\n5\t";
 8
 9    std::cout << "Eingabe: \"" << input << "\"\n" << std::endl;
10
11    Scanner scanner(input);
12
13    Token tok = scanner.getNextToken();
14
15    while (!tok.equal(TT_NIL))
16    {
17       std::cout << tok.toString() << " = " << tok.getValue() << std::endl;
18
19       tok = scanner.getNextToken();
20    }
21 }
```

Die Ausgabe sollte sein:

Zitat:

```
Eingabe: "10 + 5
* 10-
5 "

TT_NUMBER = 10
TT_PLUS = 0
TT_NUMBER = 5
TT_MUL = 0
TT_NUMBER = 10
TT_MINUS = 0
TT_NUMBER = 5
```

Man erkennt vielleicht erst hier, welchen Vorteil wir jetzt tatsächlich haben.
Der Scanner hat sich trotz der ganzen Leerräumen nicht von seiner Aufgabe abbringen lassen und konnte die Eingabe ordnungsgemäß in seine Tokens zerlegen.

Möglicherweise fragt man sich an dieser Stelle: "Wozu dieser Aufwand?"

Tatsächlich ist der Scanner ohne Parser für uns vollkommen nutzlos und bringt uns nicht weiter. Zusammen bilden sie jedoch ein perfektes Team.

Also auf zum Parser!

7 Parser
Der Scanner hat unsere Eingabe in Tokens zerlegt, was macht also nun der Parser damit?

Vereinfacht gesagt überprüft er, ob die Reihenfolge der Tokens überhaupt Sinn macht. Oder er bildet einen Abstrakten Syntaxbaum. Dazu aber später mehr, keine Sorge.

So ist die Eingabe von 5 + 5 also vollkommen korrekt, während die Eingabe von + 5 5 laut unserer Syntax keinen Sinn macht und somit ungültig ist.

Auch der Aufbau des Parsers ist sehr einfach. Er enthält als Member den Scanner und das aktuelle Token. Des Weiteren enthält er die Funktion accept und getNextToken.

Und natürlich enthält er noch die Funktionen, die wir aus der EBNF konstruieren können. Dazu eine kurze Erklärung. Man erinnere sich zurück:

Code:

```
1 [b]Start[/b] = (Multiplikation) { ("+" | "-") (Multiplikation) } ;
2
3 [b]Multiplikation[/b] = (Klammer) { ("*" | "/") (Klammer) } ;
4
5 [b]Klammer[/b] = ["+" | "-"] ((Zahl) | "(" (Start) ")") ;
6
7 [b]Zahl[/b] = (Ziffer) { (Ziffer) } ;
8
9 [b]Ziffer[/b] = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

Der Parser enthält die Funktionen start, multiplikation, klammer, zahl. Wir finden diese Namen auf der linken Seite der EBNF wieder. Das ist kein Zufall, sondern der Trick an der ganzen Sache!
Das was auf der rechten Seite steht, lässt sich dann später durch entsprechende Sprackonstrukte (Schleifen, Abfragen, etc.) realisieren. Der Rückgabewert dieser Funktionen ist ein int. Damit wird der Wert des Ausdrucks zurückgegeben.

Und schlussendlich gibt es noch die Funktion parse, die den gesamten Prozess starten soll.

Die Funktion ziffer brauchen wir allerdings nicht, da diese Aufgabe bereits durch den Scanner übernommen wurde. Er hat mehrere Ziffern zu einer Zahl zusammengefasst.

Die oben genannten Funktionen geben alle einen Integer als Wert zurück: Das Ergebnis unserer Eingabe. Auf Fließkommazahlen wurde hier verzichtet, aber wir werden das später noch ändern, versprochen.

Der Aufbau der Klasse Parser:

C++:

```
 1 class Parser
 2 {
 3     private:
 4         Scanner myScanner;
 5         Token myTok; // Zuletzt gelesenes Token
 6
 7     public:
 8         Parser(const std::string& input);
 9
10         int parse();
11
12     private:
13         int start();
14         int multiplikation();
15         int klammer();
16         int zahl();
17
18         void accept(TokenType type);
19         void getNextToken();
20 };
```

Wir beginnen mit der Besprechung der Funktion getNextToken, die einfacher nicht sein könnte:

C++:

```
void Parser::getNextToken()
{
   // Nächstes Token "abholen"
   myTok = myScanner.getNextToken();
}
```

Die Funktion accept überprüft, ob es sich um das korrekte Token handelt und gibt im Fehlerfall eine Fehlermeldung aus. Anschließend wird das nächste Token eingelesen:

C++:

```
1  void Parser::accept(TokenType type)
2  {
3     // Stimmt das Token mit dem erwarteten überein?
4     if (!myTok.equal(type))
5     {
6        std::cerr << "Error: unerwartetes Token " << myTok.toString() << ", " <<
7  TokenTypeStr[type] << " erwartet" << std::endl;
8     }
9
10    // Wir akzeptieren das Token auch im Fehlerfall, um weitere Fehler finden zu können
11    getNextToken();
   }
```

Der Konstruktor vom Parser muss, ähnlich wie der Scanner, zunächst seine Member initialisieren und das erste Token einlesen:

C++:

```
Parser::Parser(const std::string& input) : myScanner(input) // Scanner initialisieren
{
   // Erstes Token vom Scanner abholen
   getNextToken();
}
```

Die einzige Funktion, die public ist, ist die Funktion parse.

Sie startet den gesamten Parsing-Prozess:

C++:

```
1  int Parser::parse()
2  {
3     // Wir beginnen bei unserem Startsymbol Start
4     int res = start();
5
6     // Nun sollten alle Tokens abgearbeitet sein und TT_NIL das Ende der Eingabe bestätigen
7     accept(TT_NIL);
8
9     return res;
10 }
```

Man sieht hier bereits den Einsatz von accept. War die Eingabe korrekt darf nach Abarbeitung aller Tokens kein Token mehr übrig sein. Es sollte also TT_NIL zurückgegeben werden.

Die Funktion start ist Teil unserer Syntax und die Implementierung dieser Funktion hält sich an die aufgestellte EBNF:

C++:

```
// Start = (Multiplikation) { ("+" | "-") (Multiplikation) } ;
int Parser::start()
{
   // (Multiplikation)
   int res = multiplikation();

   // { ("+" | "-")
   while (myTok.equal(TT_PLUS) || myTok.equal(TT_MINUS))
   {
      switch (myTok.getType())
      {
         case TT_PLUS:
            getNextToken();
            res += multiplikation(); // (Multiplikation)
            break;

         case TT_MINUS:
            getNextToken();
            res -= multiplikation(); // (Multiplikation)
            break;
      }
   }
   // }

   return res;
}
```

Wir sehen, dass die Wiederholung (in der EBNF die geschweiften Klammern) durch eine While-Schleife realisiert ist, die solange ausgeführt wird, wie das Token TT_PLUS oder (in der EBNF das Zeichen |) TT_MINUS ist.
Und das Symbol für die Multiplikation wird durch eine weitere Funktion dargestellt, die wir einfach aufrufen.

Ein kleiner Tipp am Rande: Sollten noch Verständnisprobleme da sein, kann man den Code auch einfach um ein paar Ausgaben erweitern, die hoffentlich den Programmablauf noch verständlicher machen.

Als nächstes folgt die Implementierung der Funktion multiplikation, die vom Aufbau identisch ist:

C++:

```
// Multiplikation = (Klammer) { ("*" | "/") (Klammer) } ;
int Parser::multiplikation()
{
   int res = klammer();

   while (myTok.equal(TT_MUL) || myTok.equal(TT_DIV))
   {
      switch (myTok.getType())
      {
         case TT_MUL:
            getNextToken();
            res *= klammer();
            break;

         case TT_DIV:
            getNextToken();
            res /= klammer();
            break;
      }
   }

   return res;
}
```

Als fast letzter Schritt folgt die Funktion klammer. Zunächst der Code, auf den ich dann genauer eingehe:

C++:

```
// Klammer = ["+" | "-"] ((Zahl) | "(" (Start) ")") ;
int Parser::klammer()
{
   int sign = 1; // Für Vorzeichen (Standard: positives Vorzeichen)

   // Haben wir ein Vorzeichen?
   if (myTok.equal(TT_PLUS) || myTok.equal(TT_MINUS))
   {
      if (myTok.equal(TT_MINUS))
      {
         sign = -1; // Negatives Vorzeichen
      }

      getNextToken();
   }

   // Haben wir eine Klammer?
   if (myTok.equal(TT_LPAREN))
   {
      accept(TT_LPAREN);

      int res = sign * start(); // Rekursion

      accept(TT_RPAREN);

      return res;
   }

   // Keine Klammer, also gehen wir von einer Zahl aus
   return sign * zahl();
}
```

Wir werfen wieder einen Blick auf die EBNF und vergleichen den Quellcode damit.
Man erkennt, dass wir als erstes das optionale Vorzeichen (+ und -) bearbeiten müssen. Dazu gibt es die Variable sign, die standardmäßig von einem positiven Vorzeichen ausgeht. Sollte das Vorzeichen negativ sein wird sign auf -1 gesetzt. Anschließend überprüfen wir, ob wir eine geöffnete Klammer vorfinden. Wenn das so ist, dann rufen wir nochmals start auf.
Das Ergebnis von start wird mit der Variable sign multipliziert. Hatten wir also ein - vor der Klammer, so wird das Ergebnis der Klammer mit -1 multipliziert und wir erhalten das korrekte Ergebnis. Durch die Rekursion lassen sich auch ineinander verschachtelte Klammern korrekt auflösen.

Wenn wir keine Klammer vorgefunden haben, dann gehen wir davon aus, dass es sich um eine Zahl handeln muss und rufen die Funktion zahl auf. Die Funktion überprüft, ob es sich um eine Zahl handelt und gibt das Ergebnis zurück. Da wir auch negative Zahlen erlauben, wird auch dieses Ergebnis mit sign multipliziert. Damit sind Eingaben wie -5 möglich.

Die Implementierung der Funktion zahl:

C++:

```
// Zahl = (Ziffer) { (Ziffer) } ;
// Wir erhalten die gesamte Zahl vom Scanner:
// Es ist also keine Zusammensetzung einzelner Ziffern zu einer Zahl nötig!
int Parser::zahl()
{
   int res = myTok.getValue();

   accept(TT_NUMBER);

   return res;
}
```

Damit ist der Parser tatsächlich fertig!

Die Verwendung des Parsers könnte so aussehen:

C++:

```
 1 #include <iostream>
 2 #include "Parser.h"
 3
 4 int main()
 5 {
 6    std::string input;
 7
 8    while (true)
 9    {
10       std::cout << "Eingabe: ";
11
12       std::getline(std::cin, input);
13
14       Parser p(input);
15
16       std::cout << "Ergebnis: " << p.parse() << "\n" << std::endl;
17    }
18 }
```

Hier ein paar Beispieleingaben und die Ausgaben:

Zitat:

```
Eingabe: 10 * 5
Ergebnis: 50

Eingabe: -4 + 5
Ergebnis: 1

Eingabe: (10 + 5) - 3 * (2 - (10 / 5) * 2) - 100
Ergebnis: -79

Eingabe: ((((-100))))
Ergebnis: -100

Eingabe: ((((-100)+1)+1)+1)
Ergebnis: -97

Eingabe: ((((-100)+1)+1)+1) * (40-2)
Ergebnis: -3686

Eingabe: -((((-100)+1)+1)+1) * (40-2)
Ergebnis: 3686

Eingabe: -(-(-(+(-100)))
Error: unerwartetes Token TT_NIL, TT_RPAREN erwartet
Ergebnis: 100

Eingabe: -(-(-(+(-100))))
Ergebnis: 100
```

War doch nicht so schwer, oder? Wie wäre es, wenn unser kleines Programm in der Lage wäre die Eingabe in C++ Code zu verwandeln? Ich stelle mir das so vor:

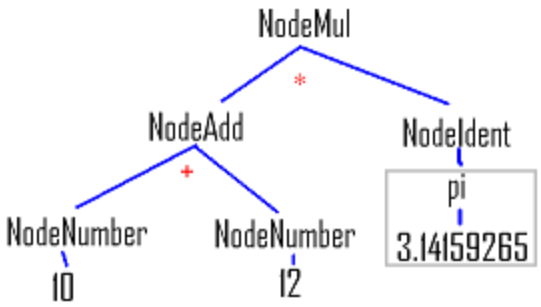
Code:

```
std::cout << [b]ADD[/b]([b]MUL[/b](5, 10), [b]SUB[/b](5, 3)) << std::endl;
```

Das macht natürlich keinen großen Sinn, aber ist eine nette Spielerei und man kann ganz genau sehen, wie die Rechnungen intern aufgelöst wurden. Außerdem möchte ich mit dieser kleinen Idee den sogenannten Abstrakten Syntaxbaum einführen.

8 Abstrakter Syntaxbaum
Was ist ein Abstrakter Syntaxbaum überhaupt? Wie der Name verrät, handelt es sich um eine baumartige Datenstruktur.

Das könnte zum Beispiel so aussehen:



Eingabe: (10 + 12) * pi

Ein solcher Abstrakter Syntaxbaum hat den Vorteil, dass wir Optimierungen vornehmen oder weitere Funktionen nachträglich einfacher hinzufügen können. Er lässt sich in C++ sehr einfach mittels Vererbung aufbauen. Allerdings müssen wir bei der Speicherreservierung und Freigabe gut aufpassen. Das wird nämlich vermehrt notwendig werden.

Wir brauchen als erstes eine abstrakte Basisklasse, die wir an dieser Stelle Node nennen:

C++:

```
1 // Basisklasse
2 class Node
3 {
4    public:
5       virtual ~Node() { }
6
7       // Ergebnis berechnen
8       virtual int eval() const = 0;
9 };
```

Die Funktion eval soll später das Ergebnis der Rechnung zurückgeben.

Die Funktion zur Codegenerierung werden wir erst später programmieren. Denn zunächst wollen wir nur, dass das Programm das gleiche macht, wie vorher. Aber eben unter Verwendung eines Abstrakten Syntaxbaums.

Jetzt benötigen wir noch eine Klasse, die eine Zahl repräsentiert und von Node abgeleitet ist. Auch das lässt sich einfach realisieren:

C++:

```
1 class NodeNumber : public Node
2 {
3    private:
4       int myVal;
5
6    public:
7       NodeNumber(int val) : myVal(val) { }
8
```

```
 9       virtual int eval() const { return myVal; }
10 };
```

Zur Realisierung der Rechenoperationen brauchen wir jetzt noch 4 weitere Klassen: NodeAdd, NodeSub, NodeMul und NodeDiv.

Ich gebe an dieser Stelle nur NodeAdd an, da die anderen 3 Klassen einen identischen Aufbau haben und sich nur in der Rechnung unterscheiden:

C++:

```
 1 class NodeAdd : public Node
 2 {
 3    private:
 4       Node *myNodeLeft;
 5       Node *myNodeRight;
 6
 7    public:
 8       NodeAdd(Node *left, Node *right) : myNodeLeft(left), myNodeRight(right) { }
 9
10       virtual ~NodeAdd()
11       {
12          delete myNodeLeft;
13          delete myNodeRight;
14       }
15
16       virtual int eval() const { return myNodeLeft->eval() + myNodeRight->eval(); }
17 };
```

Man kann das auch exemplarisch austesten, wie das später funktionieren soll:

C++:

```
Node *node = new NodeAdd(new NodeAdd(new NodeNumber(5), new NodeNumber(10)), new
NodeNumber(10)); // Speicher anfordern ...

std::cout << node->eval() << std::endl;

delete node; // ... und wieder freigeben!
```

Schreibt man den Quellcode etwas einfacher, dann würde es so aussehen und man erkennt das Prinzip des Aufbaus:

Code:

```
[b]ADD[/b]([b]ADD[/b](5, 10), 10)
```

Die Rechnung geht dabei von innen nach außen und stellt diese Rechenaufgabe dar:

Code:

```
(5 + 10) + 10
```

Der Baum soll von unserem Parser natürlich selber aufgebaut werden. Wir müssen dazu leider einige Änderungen an unserem Parser vornehmen, aber es lohnt sich!

So ändern sich natürlich die Rückgabewerte der Funktionen. Und auch die Implementierung ändert sich. Beispielsweise sieht die Funktion parse dann so aus:

C++:

```
 1 Node *Parser::parse()
 2 {
 3    // Wir beginnen bei unserem Startsymbol Start
 4    Node *res = start();
```

```
 5
 6    // Nun sollten alle Tokens abgearbeitet sein und TT_NIL das Ende der Eingabe bestätigen
 7    accept(TT_NIL);
 8
 9    return res;
10 }
```

Die Funktion start hat sich auch verändert. Wir rechnen das Ergebnis nun nicht mehr direkt aus, sondern legen mit new die entsprechenden Klassen an, die das Ergebnis durch den Aufruf von eval ausrechnen:

C++:

```
 1 // Start = (Multiplikation) { ("+" | "-") (Multiplikation) } ;
 2 Node *Parser::start()
 3 {
 4    Node *res = multiplikation();
 5
 6    while (myTok.equal(TT_PLUS) || myTok.equal(TT_MINUS))
 7    {
 8       switch (myTok.getType())
 9       {
10          case TT_PLUS:
11             getNextToken();
12             res = new NodeAdd(res, multiplikation());
13             break;
14
15          case TT_MINUS:
16             getNextToken();
17             res = new NodeSub(res, multiplikation());
18             break;
19       }
20    }
21
22    return res;
23 }
```

Die Funktion multiplikation gebe ich hier nicht an, da sie fast identisch aufgebaut ist.

Die Funktionen klammer und zahl haben sich wie folgt geändert:

C++:

```
 1 // Klammer = ["+" | "-"] ((Zahl) | "(" (Start) ")") ;
 2 Node *Parser::klammer()
 3 {
 4    int sign = 1; // Für Vorzeichen (Standard: positives Vorzeichen)
 5
 6    // Haben wir ein Vorzeichen?
 7    if (myTok.equal(TT_PLUS) || myTok.equal(TT_MINUS))
 8    {
 9       if (myTok.equal(TT_MINUS))
10       {
11          sign = -1; // Negatives Vorzeichen
12       }
13
14       getNextToken();
15    }
16
17    // Haben wir eine Klammer?
18    if (myTok.equal(TT_LPAREN))
19    {
20       accept(TT_LPAREN);
21
22       Node *res = new NodeMul(new NodeNumber(sign), start()); // Rekursion
```

```
23
24        accept(TT_RPAREN);
25
26        return res;
27    }
28
29    // Keine Klammer, also gehen wir von einer Zahl aus
30    return new NodeMul(new NodeNumber(sign), zahl());
31 }
32
33 // Zahl = (Ziffer) { (Ziffer) } ;
34 // Wir erhalten die gesamte Zahl vom Scanner:
35 // Es ist also keine Zusammensetzung einzelner Ziffern zu einer Zahl nötig!
36 NodeNumber *Parser::zahl()
37 {
38    int res = myTok.getValue();
39
40    accept(TT_NUMBER);
41
42    return new NodeNumber(res);
43 }
```

Man sieht, wir haben die Multiplikation mit der Variable sign (für Vorzeichen) unter Verwendung der Klasse NodeMul und NodeNumber realisiert. Man könnte an dieser Stelle abfragen, ob sign positiv ist und diese Multiplikation weglassen, da sie überflüssig ist, wenn sign 1 ist. Aber das lösen wir später anders, um eine mögliche Optimierung bei der Codegenerierung zu demonstrieren.

Jedenfalls haben wir damit den Parser auf einen Abstrakten Syntaxbaum umgestellt. Wir müssen natürlich noch eine letzte Änderung in der Hauptfunktion vornehmen:

C++:

```
1 // ...
2
3 // Baum generieren (Achtung: Speicher freigeben nicht vergessen!)
4 Node *node = p.parse();
5
6 std::cout << "Ergebnis: " << node->eval() << "\n" << std::endl;
7
8 // Speicher freigeben
9 delete node;
10
11 // ...
```

Wir haben daraus noch keinen Vorteil gewonnen, aber das soll sich jetzt ändern.

Geplant war ja, dass wir die Möglichkeit bieten, C++ Code zu generieren. Dazu erweitern wir unsere Basisklasse Node um folgende Funktion:

C++:

```
virtual void generateCode(std::stringstream& strm) = 0;
```

Diese Funktion muss nun von allen abgeleiteten Klassen implementiert werden. Und das ist keine große Sache.

Für die Klasse NodeNumber:

C++:

```
virtual void generateCode(std::stringstream& strm)
{
   strm << myVal;
}
```

Für die Klasse NodeAdd:

C++:

```
virtual void generateCode(std::stringstream& strm)
{
   strm << "ADD( ";
   myNodeLeft->generateCode(strm);
   strm << ", ";
   myNodeRight->generateCode(strm);
   strm << " )";
}
```

Man sieht, wir rufen für die entsprechenden Nodes auch die Funktion generateCode auf, die sie wieder für ihre Nodes aufrufen. Dadurch wird immer weiter an strm angehangen bis alles abgearbeitet wurde.

Die Funktion generateCode sieht für die Klassen NodeSub, NodeMul und NodeDiv genau so aus, nur mit dem Unterschied, dass dort **SUB**, **MUL** bzw. **DIV** verwendet wird.

Damit wir auch einen gültigen Quellcode erhalten, müssen wir uns noch um das Drumherum kümmern. Da die Funktion generateCode immer anfügt, müssen wir sowohl vor dem Aufruf als auch nach dem Aufruf, die benötigen Zeilen anfügen, um ein gültiges C++ Programm zu erhalten.

Wir lagern diesen Teil in einer eigenen Funktion aus:

C++:

```
void generateCode(Node *node, std::stringstream& strm)
{
   // Codegerüst (anfang)
   strm << "#include <iostream>\n\n"

        << "#define ADD(x, y) ((x) + (y))\n"
        << "#define MUL(x, y) ((x) * (y))\n"
        << "#define SUB(x, y) ((x) - (y))\n"
        << "#define DIV(x, y) ((x) / (y))\n\n"

        << "int main()\n"
        << "{\n"
        << "\tint res = ";

   // Den generierten Code vom Parser anhängen
   node->generateCode(strm);

   // Codegerüst (ende)
   strm << " ;\n\n"
        << "\tstd::cout << res << std::endl;\n"
        << "\treturn 0;\n"
        << "}\n";
}
```

Und die rufen wir dann ganz einfach auf:

C++:

```
Parser p(input);

// Baum generieren (Achtung: Speicher freigeben nicht vergessen!)
Node *node = p.parse();

std::cout << "\nErgebnis: " << node->eval() << "\n" << std::endl;

std::stringstream strm;

```

```
10 generateCode(node, strm); // <----------
11
12 // Speicher freigeben
13 delete node;
14
15 std::cout << strm.str() << std::endl;
```

Gibt man nun 100 + 5 * 3 - 4 ein, sollte folgender Code generiert werden:

C++:

```
 1 #include <iostream>
 2
 3 #define ADD(x, y) ((x) + (y))
 4 #define MUL(x, y) ((x) * (y))
 5 #define SUB(x, y) ((x) - (y))
 6 #define DIV(x, y) ((x) / (y))
 7
 8 int main()
 9 {
10       int res = SUB( ADD( MUL( 1, 100 ), MUL( MUL( 1, 5 ), MUL( 1,
11 3 ) ) ), MUL( 1, 4 ) ) ;
12
13       std::cout << res << std::endl;
14       return 0;
15 }
```

Hier können wir optimieren: Beispielsweise fällt die dauernde Multiplikation mit 1 auf. Sie ist durch die Multiplikation mit sign entstanden.
Wir wollen nur dann den rechten Node auswerten, falls der linke Node 1 ergibt. Dazu prüfen wir zunächst, ob es sich überhaupt um ein NodeNumber handelt. Das ist nur deshalb notwendig, da eval ansonsten den gesamten Ausdruck auswertet. Wir wollen ja später noch etwas für den C++ Compiler übrig haben. Beispiel:

Code:

```
(4 - (2+1)) * 2
```

Würden wir hier eval aufrufen, würde die gesamte linke Klammer vollständig ausgerechnet und wegoptimiert, da sie 1 ergibt. Das wollen wir aber nicht.

Wir wollen nur optimieren, falls dort bereits eine 1 steht:

Code:

```
1 * 42
```

Da wir keine Unterscheidungsmöglichkeit für die Nodes haben, nutzen wir für diese kleine Sache typeid und überprüfen so, ob es sich um eine Zahl handelt:

C++:

```
 1 virtual void generateCode(std::stringstream& strm)
 2 {
 3    if (typeid(*myNodeLeft) == typeid(NodeNumber) && myNodeLeft->eval() == 1)
 4    {
 5       myNodeRight->generateCode(strm);
 6    }
 7    else
 8    {
 9
10       strm << "MUL( ";
11       myNodeLeft->generateCode(strm);
12       strm << ", ";
13       myNodeRight->generateCode(strm);
```

```
14         strm << " )";
15     }
16 }
```

Probieren wir nun die gleiche Rechnung nochmal, erhalten wir:

C++:

```
 1 #include <iostream>
 2
 3 #define ADD(x, y) ((x) + (y))
 4 #define MUL(x, y) ((x) * (y))
 5 #define SUB(x, y) ((x) - (y))
 6 #define DIV(x, y) ((x) / (y))
 7
 8 int main()
 9 {
10         int res = SUB( ADD( 100, MUL( 5, 3 ) ), 4 ) ;
11
12         std::cout << res << std::endl;
13         return 0;
14 }
```

Man sieht, dass die unnötigen Multiplikationen weggefallen sind.

Wir wollen diesen Ausdruck als kleines Beispiel ausrechnen. Dafür beginnen wir ganz innen und bewegen uns immer weiter nach außen.

Code:

SUB(ADD(100, [b]MUL(5, 3)[/b]), 4)

Zunächst berechnen wir die Multiplikation. Das Ergebnis von 5 * 3 ist 15.

Wir setzen also 15 ein:

Code:

SUB([b]ADD(100, [u]15[/u])[/b], 4)

Die nächst innere Berechnung ist die Addition.

Wir rechnen also 100 + 15 aus und setzen wieder ein:

Code:

SUB([u]115[/u], 4)

Schlussendlich subtrahieren wir 115 von 4 und erhalten das finale Ergebnis 111.

Wenn der Parser einfach mal herausgefordert werden soll, kann man folgende Rechnung berechnen lassen.

Das Ergebnis müsste 6990 sein:

Zitat:

> (10 + 5) - 3 * (2 - (10 / 5) * 2) - 100 -((((-100)+1)+1)+1) * (40-2) + ((((-100)+1)+1)+1) - (10 + 5) - 3 * (2 - (10 / 5) * 2) - 100 -((((-100)+1)+1)+1) * (40-2) + ((((-100)+1)+1)+1)

Interessant ist natürlich auch der dafür generierte C++ Code:

Zitat:

```
int res = ADD( SUB( SUB( SUB( SUB( ADD( SUB( SUB( SUB( ADD( 10, 5 ),
MUL( 3, SUB( 2, MUL( DIV( 10, 5 ), 2 ) ) ) ), 100 ),
MUL( ADD( ADD( ADD( MUL( -1, 100 ), 1 ), 1 ), 1 ),
SUB( 40, 2 ) ) ), ADD( ADD( ADD( MUL( -1, 100 ), 1 ), 1 ),1 ) ),
ADD( 10, 5 ) ), MUL( 3, SUB( 2, MUL( DIV( 10, 5 ), 2 ) ) ) ), 100 ),
MUL(ADD( ADD( ADD( MUL( -1, 100 ), 1 ), 1 ), 1 ),
SUB( 40, 2 ) ) ), ADD( ADD( ADD(MUL( -1, 100 ), 1 ), 1 ), 1 ) ) ;
```

Damit es keine Fehler gibt, muss man immer aufpassen, dass der entstandene Code bei der Ausgabe (Konsolenfenster) nicht durch die Limitierung von Zeilenlängen ungünstig getrennt (z.B. zwischen zwei Ziffern) und damit falsch kopiert wird.

9 Endspurt
Der Matheparser funktioniert zwar, aber er ist noch etwas funktionsarm. Außerdem gibt es noch ein paar Schwächen auszubessern. Unter anderem sollen endlich Fließkommazahlen unterstützt werden. Außerdem sollen Funktionen und Konstanten ermöglicht werden.

Wir werden dazu große Teile des Codes verwerfen müssen, aber das ist es natürlich mal wieder wert. Auch ist es ein Beweis dafür, dass man ganz am Anfang alles einplanen sollte, da es später nicht mehr so einfach möglich ist.

Wir beginnen also wieder ganz vorne mit einer neuen EBNF:

Code:

```
 1 [b]Start[/b] = (Multiplikation) { ("+" | "-") (Multiplikation) } ;
 2
 3 [b]Multiplikation[/b] = (Klammer) { ("*" | "/") (Klammer) } ;
 4
 5 [b]Klammer[/b] = ["+" | "-"] ([b](Funktion) | (Bezeichner)[/b] | (Zahl) | "(" (Start) ")") ;
 6
 7 [b]Funktion[/b] = (Bezeichner) '(' [ (Start) { ',' (Start) } ] ')' ;
 8
 9 [b]Bezeichner[/b] = ( ('a' - 'Z') | '_' ) { ('a' - 'Z') | '_' } ;
10
11 [b]Zahl[/b] = (Ziffer) [ '.' ] { (Ziffer) } ;
12
13 [b]Ziffer[/b] = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

Diese EBNF ähnelt offensichtlich der am Anfang aufgestellten, wurde jedoch um Konstanten und Funktionen erweitert. Außerdem werden Fließkommazahlen unterstützt. Ein Bezeichner besteht aus einem Zeichen, gefolgt von beliebig vielen weiteren Zeichen oder einem Unterstrich.

Wir stoßen jedoch auf ein Problem. Nehmen wir folgende Eingaben:

Code:

```
PI + 10
```

Code:

```
sin(10)
```

Beim Parsen der ersten Eingabe, weiß der Parser nicht, ob nach dem Bezeichner PI noch eine öffnende Klammer kommt, wie in der zweiten Eingabe.

Das ist nicht gut, aber ich erwähnte ja anfangs bereits, dass man den Scanner auch anders konzipieren kann. Wir wollen die Gelegenheit nutzen und ihn für diese Situation (um)rüsten.

Die Idee ist zunächst folgende: Wir füllen einen std::vector mit allen Tokens. Wir können uns dann frei in dieser Liste bewegen und so entscheiden, wie das folgende Token aussieht.

Wir benennen unsere alte Funktion getNextToken zunächst in readNextToken um, und machen sie private.

Sie wird später nur noch zur Füllung der Liste genutzt.

Dann erstellen wir eine neue Funktion getNextToken mit einem Parameter peek der den Standardwert false erhält.

Außerdem benötigen wir den besagten Vector und eine Positionsangabe, damit wir wissen, wo wir uns befinden.

Das sollte dann insgesamt so aussehen:

C++:

```
 1 typedef std::vector<Token> TokenVec;
 2
 3 class Scanner
 4 {
 5    private:
 6       std::string myInput;
 7       unsigned int myPos; // Aktuelle Position in Eingabe
 8       unsigned int myVecPos;
 9       char myCh; // Aktuelles Zeichen
10
11       TokenVec myTokenVec; // Der Vector für die Tokens
12
13    public:
14       Scanner(const std::string& input);
15
16       Token getNextToken(bool peek = false);
17
18    private:
19       void initTokenVec();
20       Token readNextToken();
21       void skipSpaces();
22       void readNextChar();
23 };
```

Damit haben wir später die Möglichkeit abzufragen, welches Token folgt, ohne dabei in unserem Vector tatsächlich weiterzugehen. So können wir beim Parsen bequem entscheiden, ob es sich um eine Funktion oder eine Konstante handelt und das Problem beheben.

Natürlich müssen wir auch noch unsere Aufzählung von Tokens und so weiter ändern. Den Quellcode gebe ich dafür nicht an, aber wir brauchen ein neues Token für Bezeichner TT_IDENT und für Komma TT_COMMA.

Im Scanner sieht die Funktion getNextToken dann nur noch so aus:

C++:

```
 1 Token Scanner::getNextToken(bool peek)
 2 {
 3    // Sind wir noch nicht am Ende der Liste?
 4    if (myVecPos < myTokenVec.size())
 5    {
 6       if (!peek)
 7       {
 8          return myTokenVec[myVecPos++];
 9       }
10
11       return myTokenVec[myVecPos];
12    }
13
14    return Token(TT_NIL);
15 }
```

Man sieht, wir verändern myVecPos nicht, wenn peek true ist. Zur Füllung des Vectors schreiben wir uns noch eine Funktion, die im Konstruktor aufgerufen wird. Außerdem initialisieren wir natürlich die Variablen:

C++:

```
Scanner::Scanner(const std::string& input) : myInput(input), myPos(0), myVecPos(0) //
Initialisierungen
{
   // Erstes Zeichen einlesen
   readNextChar();

   // Alle Tokens einlesen und im Vector speichern
   initTokenVec();
}

void Scanner::initTokenVec()
{
   Token tok = readNextToken();

   while (!tok.equal(TT_NIL))
   {
      myTokenVec.push_back(tok);
      tok = readNextToken();
   }
}
```

Die größte Veränderung beim Erkennen der Tokens ist hier zu sehen:

C++:

```
Token Scanner::readNextToken()
{
   // ...

   switch (myCh)
   {
      // ...

      case ',':
         readNextChar();
         return Token(TT_COMMA);

      case 'a': case 'b': case 'c': case 'd': case 'e':
      case 'f': case 'g': case 'h': case 'i': case 'j':
      case 'k': case 'l': case 'm': case 'n': case 'o':
      case 'p': case 'q': case 'r': case 's': case 't':
      case 'u': case 'v': case 'w': case 'x': case 'y':
      case 'z':
      case 'A': case 'B': case 'C': case 'D': case 'E':
      case 'F': case 'G': case 'H': case 'I': case 'J':
      case 'K': case 'L': case 'M': case 'N': case 'O':
      case 'P': case 'Q': case 'R': case 'S': case 'T':
      case 'U': case 'V': case 'W': case 'X': case 'Y':
      case 'Z':
      case '_':
         while (isalpha(myCh) || myCh == '_')
         {
            buf += myCh;
            readNextChar();
         }

         return Token(TT_IDENT, 0, buf);

      case '0': case '1': case '2': case '3': case '4':
      case '5': case '6': case '7': case '8': case '9':
         // Einlesen, solange es eine Ziffer und kein Dezimalpunkt ist
         while (isdigit(myCh) && myCh != '.')
         {
            buf += myCh;
```

```
40            readNextChar();
41         }
42
43         // Dezimalzahl?
44         if (myCh == '.')
45         {
46            buf += myCh;
47            readNextChar();
48
49            if (!isdigit(myCh))
50            {
51               std::cerr << "Error: die Dezimalzahl ist fehlerhaft" << std::endl;
52            }
53            else
54            {
55               while (isdigit(myCh))
56               {
57                  buf += myCh;
58                  readNextChar();
59               }
60            }
61         }
62
63         // Der Einfachheit haben wir die Zahl zunächst in einen String eingelesen
64         // Dieser wird nun in ein Double konvertiert
65         return Token(TT_NUMBER, atof(buf.c_str()));
66
67      // ...
68   }
69
70   return Token(TT_NIL);
71 }
```

Damit haben wir einen Scanner, der viel flexibler ist. Wir werden beim Parsen direkt einen Abstrakten Syntaxbaum nutzen, damit wir nicht später wieder alles ändern müssen. Daher hier die dazugekommenen Klassen:

C++:

```
 1 class NodeIdent : public Node
 2 {
 3    private:
 4       std::string myName;
 5
 6    public:
 7       NodeIdent(const std::string& name) : myName(name) { }
 8
 9       const std::string& getName() const { return myName; }
10
11       virtual double eval() const
12       {
13          if (myName == "pi")
14          {
15             return 3.14159265;
16          }
17          else if (myName == "c")
18          {
19             return 299792458;
20          }
21          else if (myName == "e")
22          {
23             return 2.71828183;
24          }
25
26          std::cerr << "Error: unbekannte Konstante '" << myName << "'" << std::endl;
```

```
27
28          return 1;
29      }
30 };
```

Wir überprüfen erst beim Aufruf von eval, ob die Konstante existiert und welchen Wert sie hat. Im Fehlerfall geben wir eine Meldung aus und 1 zurück. Außerdem besitzt die Klasse eine Funktion getName, mit der wir auf den std::string zugreifen können.

Die Klasse NodeFunc ist etwas größer, aber die Erklärung folgt direkt nach dem Code:

C++:

```
 1 typedef std::vector<Node *> ArgVec;
 2
 3 class NodeFunc : public Node
 4 {
 5    private:
 6       NodeIdent *myIdent;
 7       ArgVec myArgs;
 8
 9    public:
10       NodeFunc(NodeIdent *ident) : myIdent(ident) { }
11
12       virtual ~NodeFunc()
13       {
14          for (ArgVec::iterator it = myArgs.begin(); it != myArgs.end(); ++it)
15          {
16             delete *it;
17          }
18
19          delete myIdent;
20       }
21
22       void pushArg(Node *node) { myArgs.push_back(node); }
23
24       bool checkArgCount(unsigned int count) const
25       {
26          if (myArgs.size() != count)
27          {
28             std::cerr << "Error: " << count << " Parameter fuer '" << myIdent->getName()
29 << "' erwartet" << std::endl;
30
31             return false;
32          }
33
34          return true;
35       }
36
37       virtual double eval() const
38       {
39          if (myIdent->getName() == "sin")
40          {
41             if (checkArgCount(1))
42             {
43                return sin(myArgs[0]->eval());
44             }
45          }
46          else if (myIdent->getName() == "cos")
47          {
48             if (checkArgCount(1))
49             {
50                return cos(myArgs[0]->eval());
51             }
```

```
            }
52          else if (myIdent->getName() == "tan")
53          {
54             if (checkArgCount(1))
55             {
56                return tan(myArgs[0]->eval());
57             }
58          }
59          else if (myIdent->getName() == "sqrt")
60          {
61             if (checkArgCount(1))
62             {
63                return sqrt(myArgs[0]->eval());
64             }
65          }
66          else if (myIdent->getName() == "pow")
67          {
68             if (checkArgCount(2))
69             {
70                return pow(myArgs[0]->eval(), myArgs[1]->eval());
71             }
72          }
73          else
74          {
75             std::cerr << "Error: unbekannte Funktion '" << myIdent->getName() << "'" <<
76 std::endl;
77          }
78
79          return 1;
80       }
    };
```

Zunächst benötigen wir einen std::vector damit wir die Parameter der Funktion speichern können. Dieser Vector wird während des Parsings mit der Funktion pushArg befüllt. Außerdem gibt es noch eine Funktion checkArgCount, die überprüft, ob der Vector die angegebene Anzahl von Parametern enthält. Im Fehlerfall wird eine Fehlermeldung ausgegeben.

Die Funktion eval überprüft dann einfach den Namen der Funktion und ob die entsprechende Anzahl von Parametern vorliegt. Ist das alles okay, wird die Funktion aufgerufen und das Ergebnis zurückgegeben. Ansonsten wird auch an dieser Stelle wieder 1 zurückgegeben und eine Fehlermeldung ausgegeben.

Nun zum Parser: Durch die neue EBNF erkennen wir, dass wir zwei neue Funktionen benötigen. Und zwar die Funktion funktion und bezeichner.

Die Funktion bezeichner ist wie die Funktion zahl sehr einfach:

C++:

```
 1 // Bezeichner = ( ('a' - 'Z') | '_' ) { ('a' - 'Z') | '_' } ;
 2 // Wir erhalten den gesamten Bezeichner vom Scanner:
 3 // Es ist hier also keine Zusammensetzung mehr notwendig!
 4 NodeIdent *Parser::bezeichner()
 5 {
 6    std::string ident = myTok.getStrValue();
 7
 8    accept(TT_IDENT);
 9
10    return new NodeIdent(ident);
11 }
```

Die Funktion funktion enthält aber auch nicht viel neues:

C++:

```
1 // Funktion = (Bezeichner) '(' [ (Start) { ',' (Start) } ] ')' ;
2 NodeFunc *Parser::funktion()
3 {
4    NodeFunc *res = new NodeFunc(bezeichner());
5
6    accept(TT_LPAREN);
7
8    if (!myTok.equal(TT_RPAREN))
9    {
10      res->pushArg(start()); // Rekursion
11
12      while (myTok.equal(TT_COMMA))
13      {
14         accept(TT_COMMA);
15         res->pushArg(start()); // Rekursion
16      }
17   }
18
19   accept(TT_RPAREN);
20
21   return res;
22 }
```

Man beachte, dass der Konstruktor von NodeFunc die Funktion bezeichner aufruft. Des Weiteren wird mit der Funktion pushArg der Vector gefüllt, falls Parameter vorhanden sind.

Eine weitere Änderung erfolgt in der Funktion klammer:

C++:

```
1 // Klammer = ["+" | "-"] ((Funktion) | (Bezeichner) | (Zahl) | "(" (Start) ")") ;
2 Node *Parser::klammer()
3 {
4    int sign = 1; // Für Vorzeichen (Standard: positives Vorzeichen)
5
6    // Haben wir ein Vorzeichen?
7    if (myTok.equal(TT_PLUS) || myTok.equal(TT_MINUS))
8    {
9       if (myTok.equal(TT_MINUS))
10      {
11         sign = -1; // Negatives Vorzeichen
12      }
13
14      getNextToken();
15   }
16
17   // Haben wir eine Klammer?
18   if (myTok.equal(TT_LPAREN))
19   {
20      accept(TT_LPAREN);
21
22      Node *res = start(); // Rekursion
23
24      accept(TT_RPAREN);
25
26      return new NodeMul(new NodeNumber(sign), res);
27   }
28   else if (myTok.equal(TT_IDENT))
29   {
30      // Ist das folgende Token eine öffnende Klammer?
31      if (myScanner.getNextToken(true).equal(TT_LPAREN))
32      {
33         // Dann ist es eine Funktion
34         return funktion();
```

```
35        }
36
37        // Ansonsten eine Konstante
38        return bezeichner();
39    }
40
41    // Keine Klammer, also gehen wir von einer Zahl aus
42    return new NodeMul(new NodeNumber(sign), zahl());
43 }
```

Wenn keine öffnende Klammer gefunden wurde, wird überprüft, ob es sich um einen Bezeichner handelt. Ist das der Fall, dann müssen wir entscheiden, ob dieser Bezeichner eine Funktion beschreibt, also noch eine öffnende Klammer folgt oder ob es sich um eine Konstante handelt, also **keine** öffnende Klammer folgt.

Damit wir das überprüfen können, übergeben wir getNextToken den Parameter true. So überprüfen wir das folgende Token ohne es tatsächlich zu lesen und die Positionsangabe im Scanner zu ändern. Das war unser Ziel, als wir den Scanner umgerüstet haben.

Wenn wir also eine Klammer vorfinden, handelt es sich um eine Funktion. Ansonsten handelt es sich um einen Bezeichner. Wir geben das Ergebnis dieser Funktion natürlich auch mit return zurück.

Damit haben wir eigentlich alles geschafft, was wir schaffen wollten. Man kann jetzt noch weiter machen und Fehlerabfragen einprogrammieren, ob z.B. eine Division durch 0 stattfindet. Dazu müsste man die eval Funktion von NodeDiv verändern:

C++:

```
 1 virtual double eval() const
 2 {
 3    double right = myNodeRight->eval();
 4
 5    // Rechter Ausdruck ergibt nicht 0
 6    if (right != 0)
 7    {
 8       return myNodeLeft->eval() / right;
 9    }
10
11    std::cerr << "Error: Division durch 0" << std::endl;
12
13    return 1;
14 }
```

Hier mal wieder ein paar Beispieleingaben und ihre Ausgaben:

Zitat:

```
Eingabe: pi
Ergebnis: 3.14159

Eingabe: e
Ergebnis: 2.71828

Eingabe: e + pi * pow(2, 2)
Ergebnis: 15.2847

Eingabe: pow(pow(pow(2, 4), 2), pi)
Ergebnis: 3.67882e+007

Eingabe: pow ( sin ( 4 - pi) * pow(3 + 1, 4 - 0), 1)
Ergebnis: 193.741

Eingabe: sin() + pow()
```

```
Error: 1 Parameter fuer 'sin' erwartet
Error: 2 Parameter fuer 'pow' erwartet
Ergebnis: 2

Eingabe: pow(1, 2)
Ergebnis: 1

Eingabe: test * pi
Error: unbekannte Konstante 'test'
Ergebnis: 3.14159

Eingabe: test() * pi
Error: unbekannte Funktion 'test'
Ergebnis: 3.14159
```

Auch verschachtelte Funktionsaufrufe stellen kein Problem dar. Auch Leerzeichen und andere Leerräume bereiten noch immer keine Probleme.

10 Abschluss
Vielleicht hat ja schon jemand versucht, selbst einen solchen Matheparser zu entwickeln und hat einfach eine eigene Idee genutzt. War dieser auch so mächtig oder stieß man schnell an die Grenzen? Wenn man nach der hier vorgeschriebenen Idee vorgeht, dann kann man sehr robuste Programme schreiben. Dabei ist es egal, ob es sich um eine Syntax handelt, die die Mathematik beschreibt, oder um eine Syntax, die eine kleine Scriptsprache darstellt. Es ist sehr einfach, wenn man sich nur etwas Zeit nimmt und die EBNF versucht zu verstehen und anzuwenden.

Und wenn man sich den Code ansieht, dann stellt man fest, dass dieser nicht sehr umfangreich ist. Er würde sich sogar noch kürzen lassen, wenn man beim Abstrakten Syntaxbaum nicht für jede Rechenoperation einen eigenen Node erstellt sondern nur einen und stattdessen einen Parameter übergibt, welche Rechenart auszuführen ist (Im C# Programm wurde das so gelöst).

Auch könnte man für die Funktionen, wie sin, cos oder die Konstanten je einen eigenen Node erzeugen und damit die If-Abfragen vermeiden. Oder man programmiert alles so, dass es ganz einfach erweiterbar ist. Doch die zugrundeliegende Idee bleibt natürlich gleich.

Wie anfangs erwähnt, würde ich diesen Artikel gerne fortsetzen und im nächsten Teil tatsächlich auf die Entwicklung einer kleinen Scriptsprache eingehen, falls dazu das Interesse besteht. Dabei sollen die vorgestellten Ideen von Scanner, Parser, Abstraktem Syntaxbaum weiter ausgebaut und praktisch eingesetzt werden.

11 Downloads

- In diesem Artikel entwickelter Matheparser in C++
- Matheparser in C (einfache Version)
- Matheparser in C#

knivil Mitglied 09:25:21 07.06.2010 Titel:

Ich finde NodeFunc unguenstig. Warum nicht genau wie Add fuer jede Funktion eine extra Klasse wie:

C++:

```
class Cos : public Node
{
   public:
   Cos(Node* right);

      virtual double eval();
};
```

Auch wuerde ich der eval-Methode (oder dem Konstruktor) einen Kontext mitgeben, in der Variablen und Konstanten definiert sind.

Th69 Mitglied 10:52:04 07.06.2010 Titel:

Hallo Dummie,

da Compilerbau mein Spezialgebiet an der Uni war, habe ich mich auch privat damit weiter beschäftigt.

Zuerst in C++: http://www.bitel.net/dghm ..... ads/FctParser-Sources.zip
(für den Borland C++ Builder) - dieser ist auch in einigen Beiträgen hier im Forum schon verlinkt.

Und ähnlich wie du habe ich dann einen Artikel verfasst (jedoch für C#): http://www.mycsharp.de/wbb2/thread.php?threadid=71995

Mein MathParser ist jedoch so allgemein gehalten, daß man die Operatoren, Funktionen und Konstanten jeweils selber der Basisklasse übergeben kann (auch die interne Optimierung als Abstract Syntax Tree ist implementiert!), d.h. es gibt eine allgemeine NodeFunc-Klasse (welche knivil ja angemeckert hat -).

Ich finde es schön, daß du dir die Mühe für diesen Artikel gemacht hast. Und ich wäre sehr an einer Fortsetzung interessiert (ich selber hatte vor ca. 13 Jahren mal eine eigene Scriptsprache in Delphi 2 (!) implementiert)...

Marc-O Mitglied 12:38:10 07.06.2010 Titel:

Hallo Dummie

Ich finde den Artikel sehr interessant und gut beschrieben. Das ganze schaut sehr ausbaufähig aus und auch ich wäre sehr interessiert an einer Fortsetzung.

Wollt auch schon mal mit einer Scriptsprache anfangen aber ich schaff es immer nicht Projekte für die ich keinen direkten Sinn (Anwendung) finde zu Ende zu bringen :-). Deswegen großen Respekt an dir das du dir die ganze Mühe gemacht hast und auch noch einen Artikel dafür geschrieben hast.

Mfg Marco

Dummie Mitglied 14:28:58 07.06.2010 Titel:

Hallo,

danke für die positiven Kommentare

knivil schrieb:

Ich finde NodeFunc unguenstig. Warum nicht genau wie Add fuer jede Funktion eine extra Klasse wie:

C++:

```
class Cos : public Node
{
   public:
   Cos(Node* right);

       virtual double eval();
};
```

> Auch wuerde ich der eval-Methode (oder dem Konstruktor) einen Kontext mitgeben, in der Variablen und Konstanten definiert sind.

In dem Fall müsste man beim Parsing entscheiden, welcher Node angelegt werden muss. Das lässt sich sicherlich gut über ein Array o. ä. erledigen - so ließen sich auch viel einfacher neue Funktion hinzufügen. Für diesen Artikel fand ich das Vorgehen über NodeFunc aber als verständlicher.

Th69 schrieb:

> Mein MathParser ist jedoch so allgemein gehalten, daß man die Operatoren, Funktionen und Konstanten jeweils selber der Basisklasse übergeben kann (auch die interne Optimierung als Abstract Syntax Tree ist implementiert!), d.h. es gibt eine allgemeine NodeFunc-Klasse (welche knivil ja angemeckert hat -).

Dein Matheparser ist auch wirklich sehr interessant, insbesondere da er so allgemein gehalten ist. Ich hatte ja auch im Artikel beschrieben, dass man das machen kann, damit haben wir ja nun auch dafür ein schönes Beispiel.

Marc-O schrieb:

> Wollt auch schon mal mit einer Scriptsprache anfangen aber ich schaff es immer nicht Projekte für die ich keinen direkten Sinn (Anwendung) finde zu Ende zu bringen :-). Deswegen großen Respekt an dir das du dir die ganze Mühe gemacht hast und auch noch einen Artikel dafür geschrieben hast.

Ja, so eine Sprache ist auch schon ein etwas größerer Akt. Selbst eine sehr kleine Scriptsprache kann schon sehr viel Arbeit bedeuten und man stellt erst beim Programmieren fest, was moderne Sprachen tatsächlich alles für einen leisten. Das was man schon lange als selbstverständlich empfunden hat, vermisst man plötzlich bei der eigenen.

chrmhoffmann Unregistrierter 17:11:37 08.06.2010 Titel: **boost::spirit**

Hi,

ich schreibe keine parser mehr selbst. boost::spirit kann das viel besser.

Chris

milan1612 Mitglied 21:47:19 08.06.2010 Titel:

Sehr schöner Artikel und sehr interessant!

Marc-O Mitglied 06:37:28 11.06.2010 Titel: **Re: boost::spirit**

---

chrmhoffmann schrieb:

> Hi,
>
> ich schreibe keine parser mehr selbst. boost::spirit kann das viel besser.
>
> Chris

Habs mir mal angeschaut und muss sagen ist sehr interressant, wobei ich einiges aus dem Artikel direkt mitnehmen konnte für diese Bibliothek, und wenn es auch nur das EBNF ist :-)

Aber ansonst werd ich mir die mal genauer anschauen und versuchen nen kleinen XML-Parser damit zu bauen.

Mfg marco

knivil Mitglied 08:47:27 11.06.2010 Titel:

---

flex + bison

iJake1111 Unregistrierter 17:06:18 30.11.2010 Titel: **Danke! Klasse Tutorial**

Das ist wohl eines der Besten C Tutorials, das ich je gesehen hab, Respekt! Seit diesem Tut, sehe ich Parser schon mit ganz anderen Augen

Nochmal rechten Dank!
Mfg iJake1111