Programmieren / Algorithmen & Datenstrukturen

Grundlagen (i), Teil 5

Prof. Dr. Skroch

Universitatea
BABEŞ-BOLYAI

Grundlagen (i)

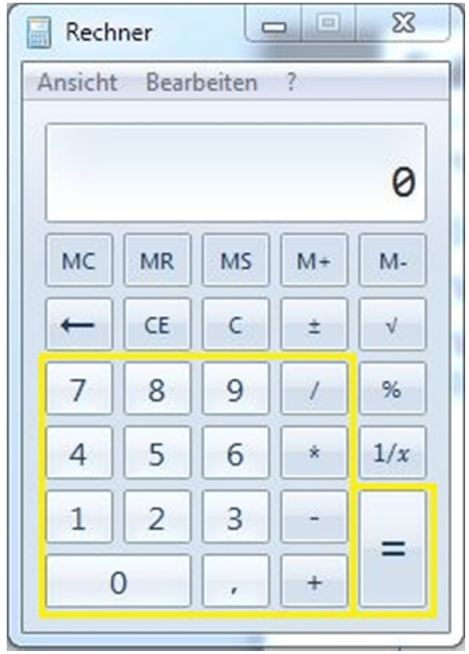
Inhalt.

- Hallo C++
- Objekte, Typen, Werte, und Steuerungsprimitive
- Berechnungen und Anweisungen
- Fehler
- Fallstudie: Taschenrechner
- Funktionen und Programmstruktur
- Klassen

Fallstudie: Taschenrechner

Die Anforderung lautet, einen einfachen Taschenrechner zu programmieren, der die vier Grundrechenarten und Klammern verarbeiten kann.

- Das Problem ist bekannt, Lösungen auch.
  - Taschenrechner existieren als Geräte.
  - Taschenrechner-Programme existieren als Mini-Anwendungen für den Desktop.
- Genauere Spezifikation unserer Anforderungen:
  - Ein- und Ausgaben über die Konsole (keine GUI).
  - Einschränkung auf die vier Grundrechenarten und Klammern.
  - Ein Rechenausdruck soll eingelesen werden.
  - Das korrekte Ergebnis der Rechnung soll ausgegeben ausgeben.
  - Wenn z.B. `(2+3.1*4)-1` eingegeben wird, soll die Ausgabe `13.4` lauten.

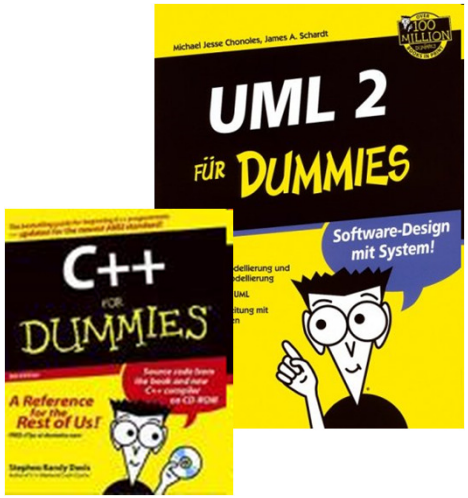


- Wie geht man vor?
  - Jedenfalls nicht die IDE starten und losschreiben `int main( ) try { ...`
  - Sondern zuerst nachdenken.

Herausforderungen

Software Engineering ist *nicht* die "Produktion" von Software, sondern die anspruchsvolle Tätigkeit der Entwicklung neuer Softwarelösungen.

- In Theorie und Praxis trifft man leider oft auf fehlendes Verständnis und mangelnde Professionalität.
  - Zum Glück findet man solche Bücher *nicht*:
    - „Motoren neu entwickeln: von der Idee zur Serienreife in drei Wochen“
    - „In fünf Tagen vom Bauantrag zum fertigen Wolkenkratzer“
    - „Ihr eigener Herzkatheder selbst gemacht: ein do-it-yourself Ratgeber für Dummies“
  - Warum aber:



Derartige Versprechungen ziehen oft Menschen an, die sich von den Anstrengungen professioneller Qualitätsarbeit befreien wollen.

Diese Zielgruppe lässt sich bereitwillig suggerieren, dass die professionelle Entwicklung von Software

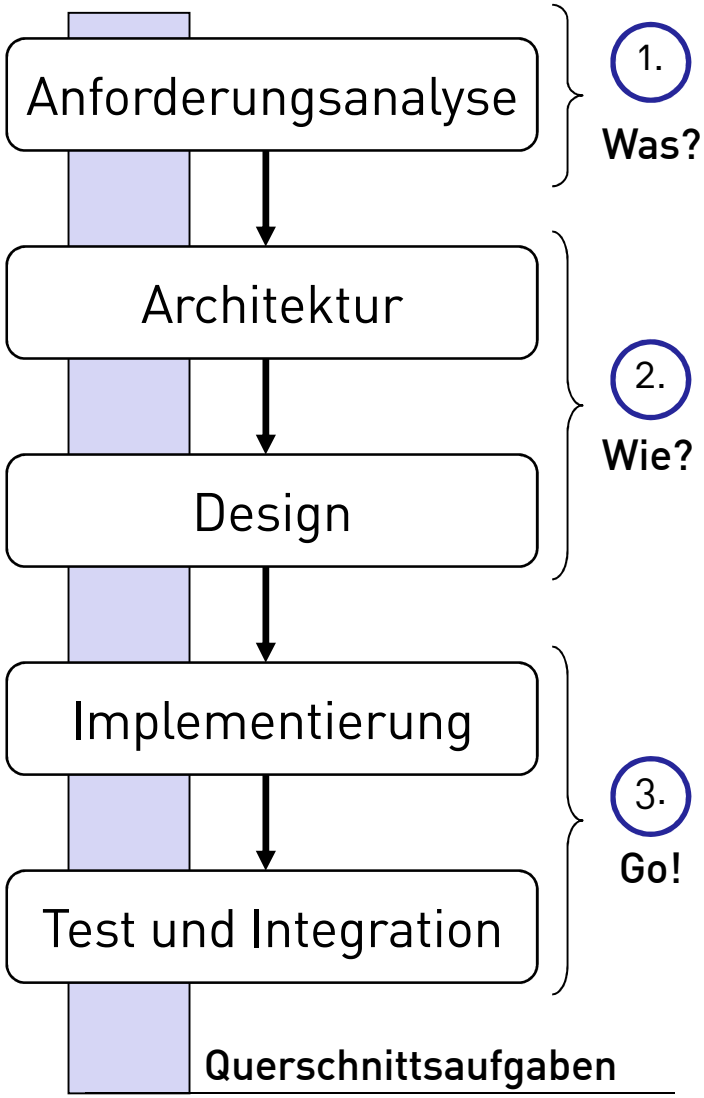
- ganz einfach sei
  (lesen Sie dazu etwa "No silver bullet" von F. Brooks)
- und mühelos von jedermann beherrscht werden könne
  (eine sorgfältige Ausbildung und die stetige Übung in der Praxis gehören zu den Voraussetzungen, um darin gut zu werden).

Bildquelle: Wiley.

Die Phasen der Softwareentwicklung

Softwareentwicklung durchläuft schrittweise und systematisch die unterschiedlichen Phasen eines gewählten Vorgehensmodells.

- Anforderungen: was soll die Software machen?
  - Problembeschreibung: „Lasten“.
- Architektur: wie soll es gehen?
  - Lösungsbeschreibung: „Pflichten“.
- Design: wie ganz genau soll es gehen?
  - Lösungsbeschreibung: „Pflichten“.
- Umsetzung der Programmiervorgaben in die Zielsprache.
- Zusammenbau der Komponenten, Nachweis, dass die Lasten erfüllt werden, Auslieferung.



Iterativ-inkrementelles Vorgehen

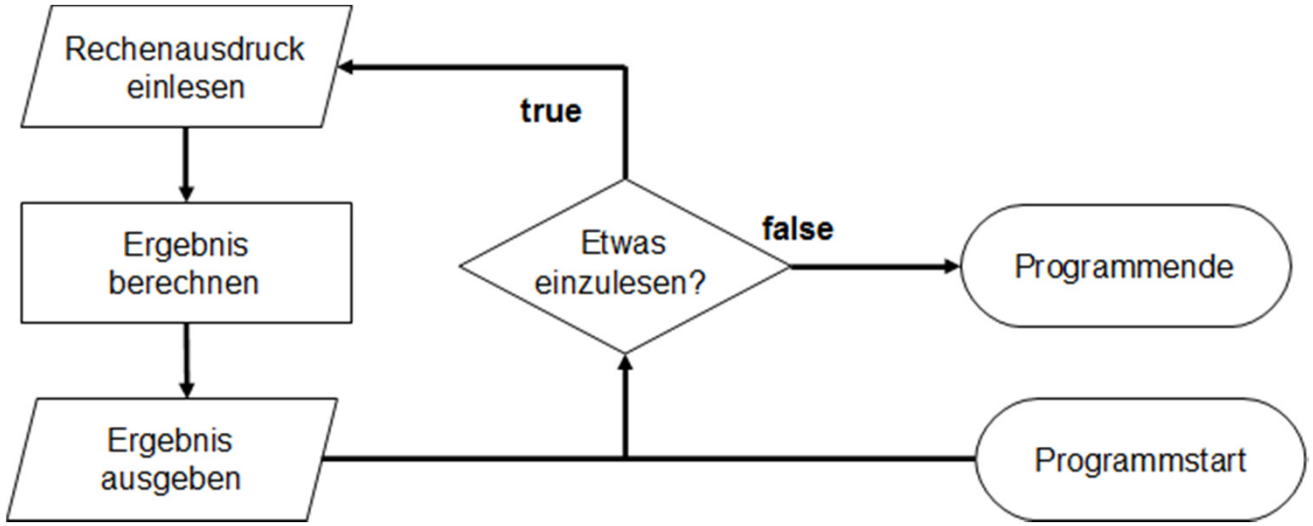
Es wird nicht das ganze Problem auf einen Streich gelöst, sondern man geht Schritt für Schritt vor.

- Inkremente: Schritt $n + 1$ soll fehlende Funktionalität aus Schritt n liefern.
  - Zerlegen Sie das Programm in möglichst kleine, logisch zusammen gehörende Teile, die ausschließlich explizite gegenseitige Abhängigkeiten haben.
  - Zerlegen Sie ggf. die Teile solange weiter in immer kleinere Teile, bis Sie die einzelnen Teile isoliert programmieren und austesten können.
  - Setzen Sie stabile Teilfunktionalitäten nach und nach (inkrementell) zur Gesamtlösung zusammen, testen Sie die Integrationsschritte.
  - Es ist meist nicht einfach, eine gute Lösung (Zerlegung des Problems) zu finden: was genau *sind* die unabhängigen Einzelteile?
- Iterationen: Schritt $n + 1$ soll offenen Probleme aus Schritt n lösen.
  - Verbessern Sie ihre Programmteile, indem Sie in Folgeversionen kontinuierlich die Fehler aus den Vorversionen eliminieren.
- Prototyp(en)
  - Erstellen Sie möglichst früh eine kleine, eingeschränkte Version des Gesamtprogramms, die lauffähig ist und einen ersten, wichtigen Teilaspekt löst.

Zurück zum Taschenrechner

Erste Überlegungen zum Mini-Rechner Programm.

- Ist die Problemstellung (Lasten) klar?
  - In realen Software-Entwicklungsprojekten (in denen Lasten oft u.a. anhand von sog. Anwendungsfällen dargestellt werden) ist sie das fast *nie*.
  - In unserer Fallstudie ist sie zum Glück zunächst klar genug.
- Erste grobe Lösungsidee:



Quellcode Version 0.0

Der erste Versuch: Quellcode für einen Mini-Rechner.

- Wir konnten uns nicht zurückhalten und haben direkt losprogrammiert… nach einiger Zeit sind wir bei ungefähr dieser Lösung angekommen:

```
int main() try {
    int lval{0}; int rval{0}; char op{' '};
    cout << "Rechenausdruck (mit +,-,*,/, Ende mit ;) "; // wie z.B. 2*3+4;
    cin >> lval;
    if( !cin ) { cerr << "Kein linker Operand"; return -1; }
    while( cin >> op  &&  op != ';' ) {
       if( !cin ) { cerr << "Kein Operator"; return -2; }
       cin >> rval;
       if( !cin ) { cerr << "Kein rechter Operand"; return -3; }
       switch( op ){
         case '+': lval += rval; break;
         case '-': lval -= rval; break;
         case '*': lval *= rval; break;
         case '/': lval /= rval; break;
         default : { cerr << "Unbekannter Operator"; return -4; }
       }//switch()
    }// while()
    cout << "Ergebnis: " << lval << endl;
    return 0;
}
// …
```

- Probieren Sie dieses Programm aus.

Quellcode Version 0.0

Die wichtigsten Erkenntnisse.

- Die Auswertung des eingegebenen Rechenausdrucks ist offenbar nicht ausreichend.
  - v0.0 kennt die Punkt-vor-Strich-Regel nicht.
    - Z.B. `1+2*3` erzeugt `9` und nicht `7`, z.B. `1*2-4/2` erzeugt `-1` und nicht `0`.
  - v0.0 erkennt keine mit Klammern verschachtelten Rechenausdrücke.
- Die Aufgabe – ein Mini-Rechner für die vier Grundrechenarten – erweist sich damit als deutlich schwieriger als vielleicht erwartet.
  - Man kann nicht einfach von links nach rechts auswerten, der Rechenausdruck muss also vorausschauend ausgewertet werden, ob z.B. nach einem + noch ein * kommt.
  - Wie kann man eine eingegebene Berechnungszeile wie `1+2*3` oder `45/(7*(2+11))` geeignet als Daten repräsentieren, oder (vielleicht besser) direkt verarbeiten?
- Es ist jetzt ein konzeptioneller Sprung erforderlich, den man kaum ohne Vorwissen und nur durch eigene Überlegungen schafft.
  - Hier zeigt sich die Bedeutung von guter Fachliteratur, und Dozenten und deren Erfahrung.
- Der heute übliche Lösungsweg führt über eine *formale Grammatik*.

Formale Grammatiken

Was ist eine Grammatik?

- Wichtiges zu Grammatiken:
  - Eine Grammatik ist eine Menge von Regeln für die Syntax eines Ausdrucks.
    - Syntax ist die formale Ordnung von Symbolen.
  - Die Regeln einer Grammatik geben vor, wie ein Ausdruck aufgebaut sein soll.
  - Den Vorgang, einen Ausdruck nach den Regeln einer formalen Grammatik zu verarbeiten, nennt man laut Duden *parsen* des Ausdrucks.
- Beispiele
  - `2*3+4/2` ist grammatisch korrekt, `2*+342/` nicht.
  - *Flugzeuge fliegen aber Züge fahren* ist grammatisch korrekt, *fliegen fahren Züge aber Flugzeuge* ist es nicht.
- Warum sind die einen Ausdrücke korrekt und die anderen nicht?
  - Woher weiß es ein Mensch?
  - Wie kann es programmiert werden?

Beispiel einer Grammatik

Eine Listen-Grammatik.

Liste :
"{" Sequenz "}"

Sequenz :
Element
Element "," Sequenz

Element :
"A"
"B"

- Gültige Listen z.B.
  - {A}
  - {B}
  - {A,B,B,B}
  - {B,A,B,B,A}
- Ungültige Listen z.B.
  - { }
  - A
  - {A,,B}
  - {A,A,B,B,A,A,B,A,A,B,B,B,B,8,B,B,A}
  - {A,A,B,B,B,A,B,}
  - {,A,A,B,A}
  - {ABBA}

Weiteres Beispiel einer Grammatik

Eine Grammatik für eine kleine Teilmenge der deutschen Sprache.

Satz :
Substantiv Verb
Satz Konjunktion Satz

Konjunktion :
"und"
"oder"
"aber"

Substantiv :
"Flugzeuge"
"Züge"
"Computer"
"Feuerwehrmänner"

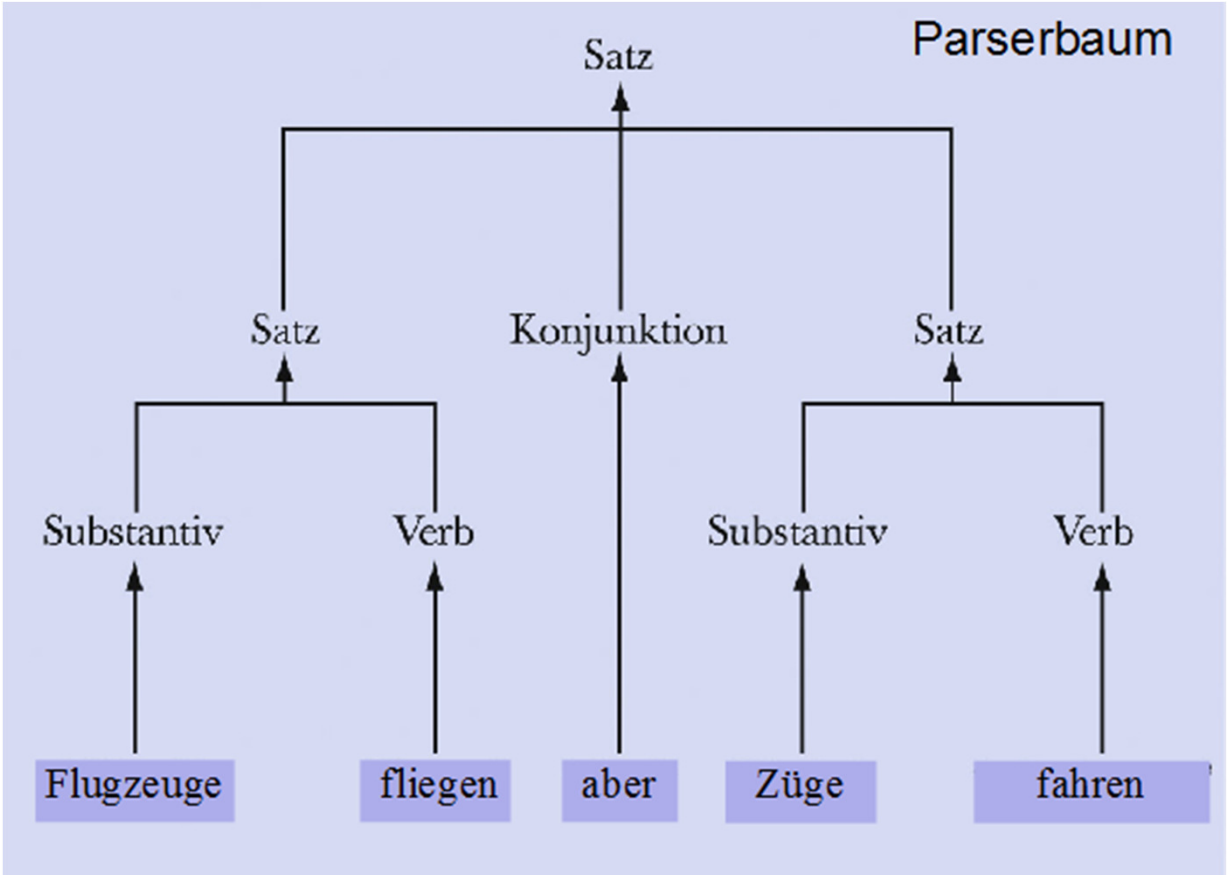
Verb :
"fliegen"
"fahren"
"rechnen"
"löschen"

- Bilden Sie einige gültige und einige ungültige Sätze mit der Grammatik und erklären Sie jeweils, warum ein Satz gültig ist oder nicht.
- Flugzeuge löschen und Computer fliegen aber Flugzeuge fahren
  ist gültig.
- Flugzeuge fliegen und Computer rechnen aber Feuerwehrmänner löschen.
  ist nicht gültig.
  - Wird durch eine Erweiterung der Grammatik gültig:

    SatzPunkt :
    Satz "."
    Satz :
    ...

Weiteres Beispiel einer Grammatik

Parser- oder Syntaxbaum.



Weiteres Beispiel einer Grammatik

Ganze Zahlen.

GanzeZahl :
PosGanzeZahl
'-' PosGanzeZahl
'0'

PosGanzeZahl :
ZifferOhneNull
ZifferOhneNull FolgeZiffern

ZifferOhneNull :
'1'
'2'
'3'
'4'
'5'
'6'
'7'
'8'
'9'

FolgeZiffern :
Ziffer
Ziffer FolgeZiffern

Ziffer :
'0'
ZifferOhneNull

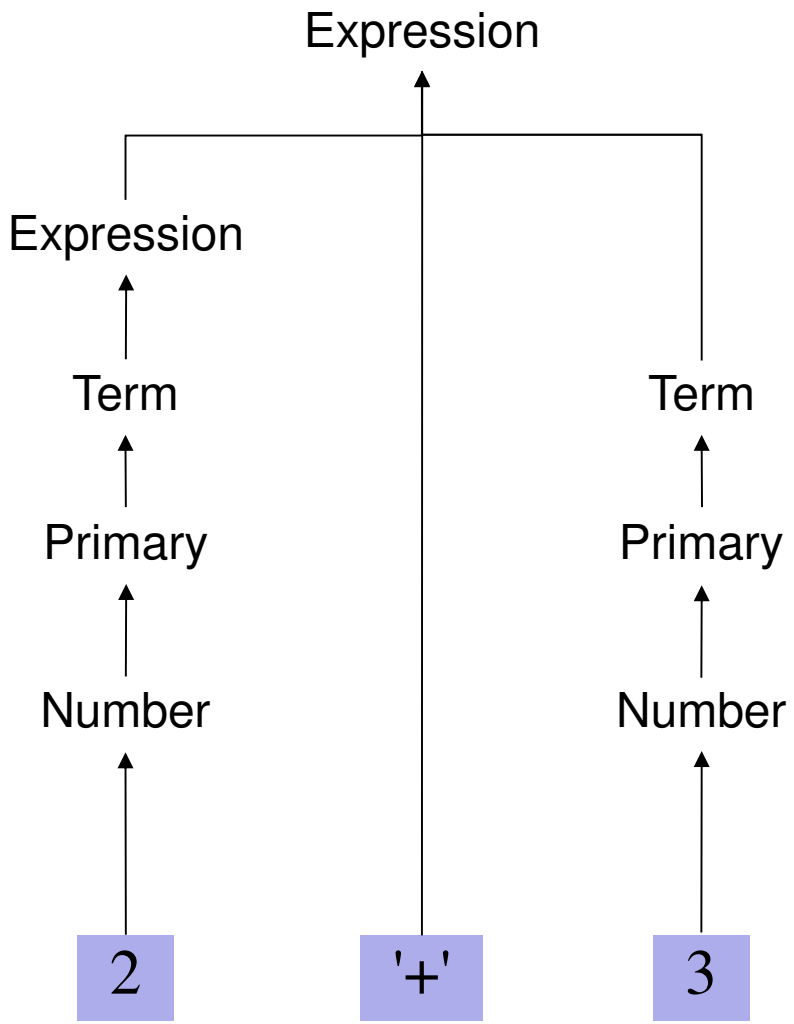
Die formale Grammatik der vier Grundrechenarten

Die formale Grammatik der Grundrechenarten definiert die Syntax für einen Rechenausdruck in einer allgemeinen, formalen und eleganten Art.

Expression :	*Rechenausdruck*
Expression '+' Term	*als "linker Strichoperand"*
Expression '-' Term	
Term	
Term :	*Summand*
Term '*' Primary	*als "linker Punktoperand"*
Term '/' Primary	
Primary	
Primary :	*Faktor*
Number	*Zahl*
'(' Expression ')'	*Rechenausdruck in Klammern*
Number :	
floating-point literal	*C++ Literal für Gleitkommawerte*

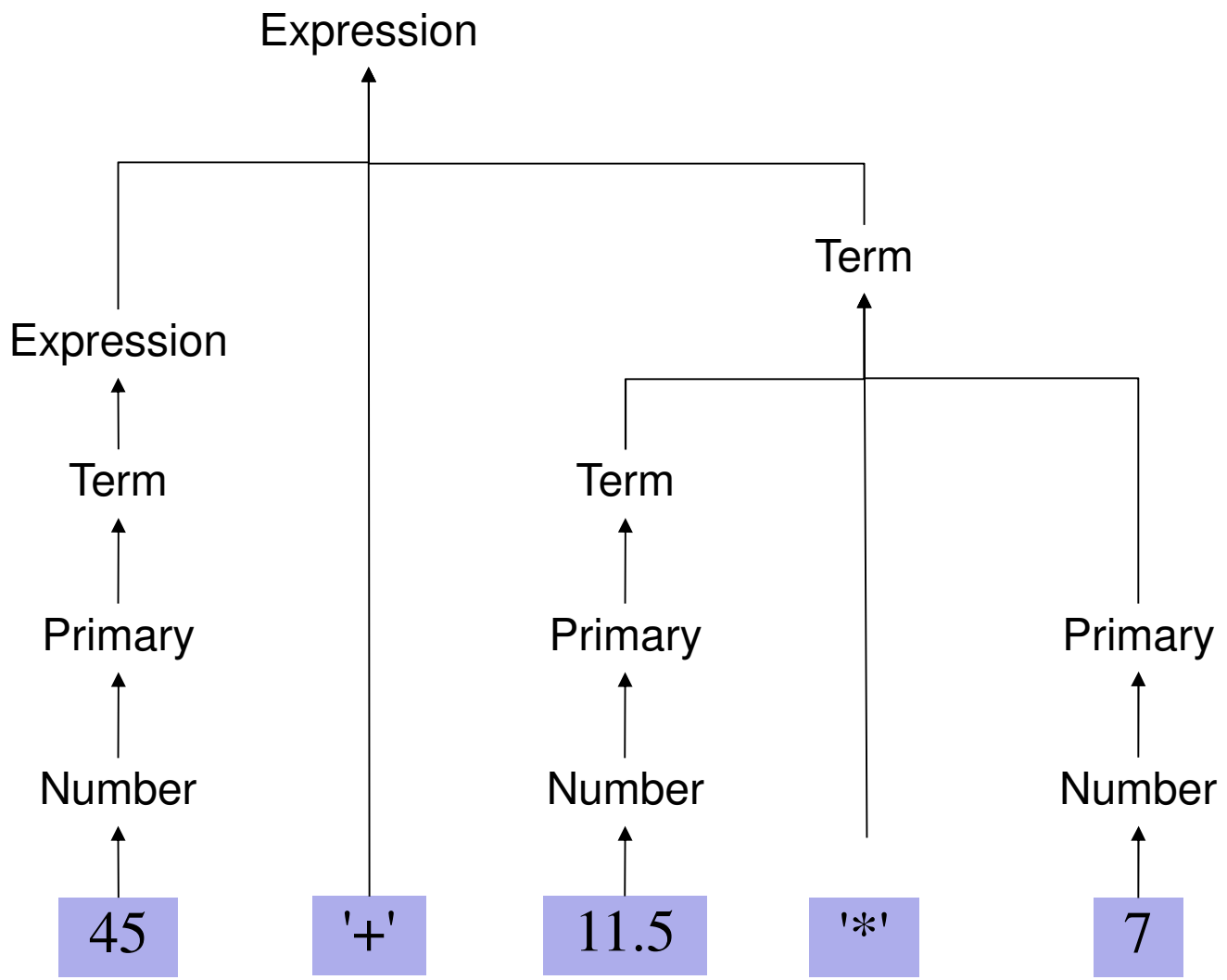
Die formale Grammatik der vier Grundrechenarten

Der Parserbaum für den Ausdruck `2+3`.



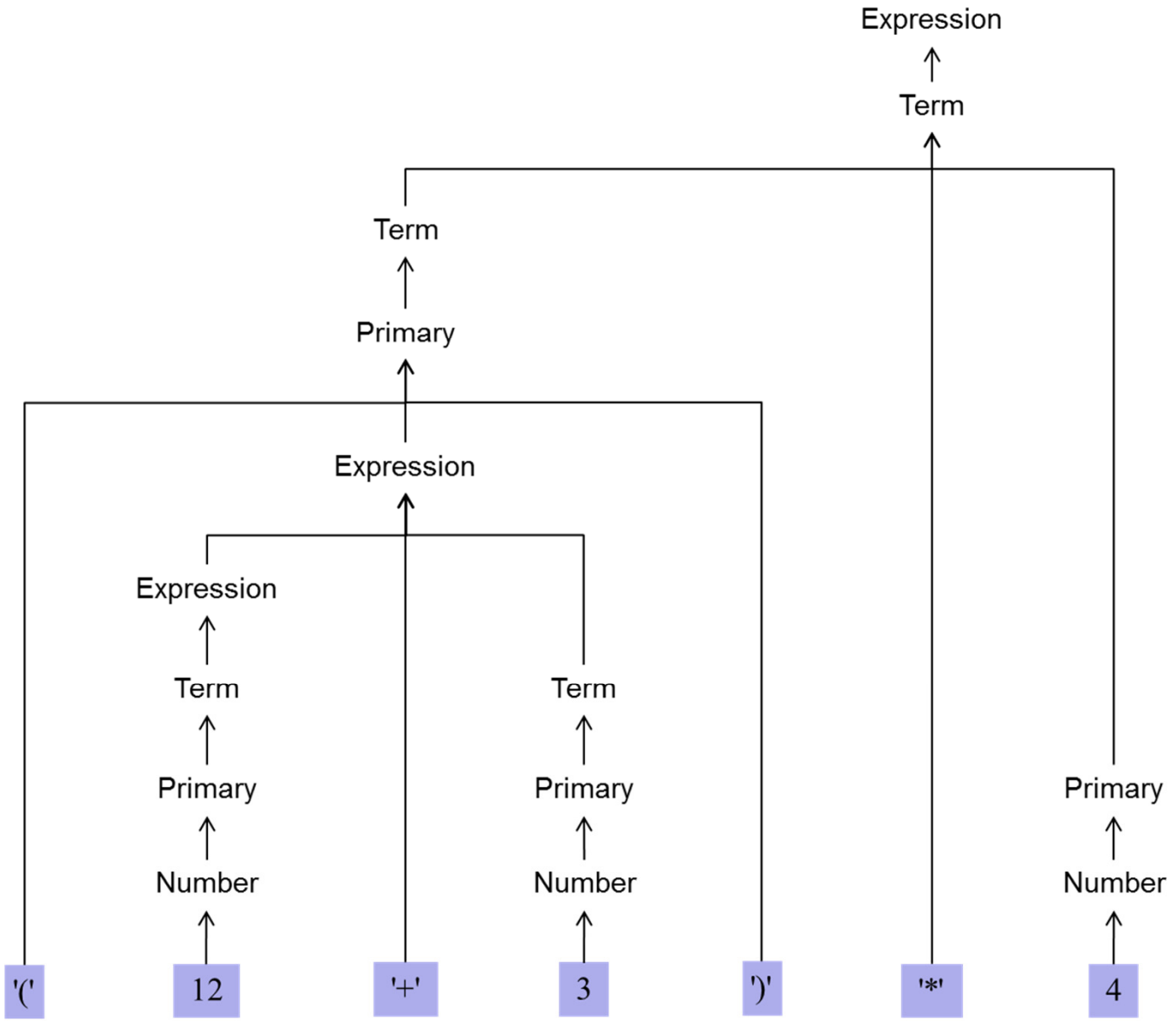
Die formale Grammatik der vier Grundrechenarten

Der Parserbaum für den Ausdruck `45+11.5*7`.



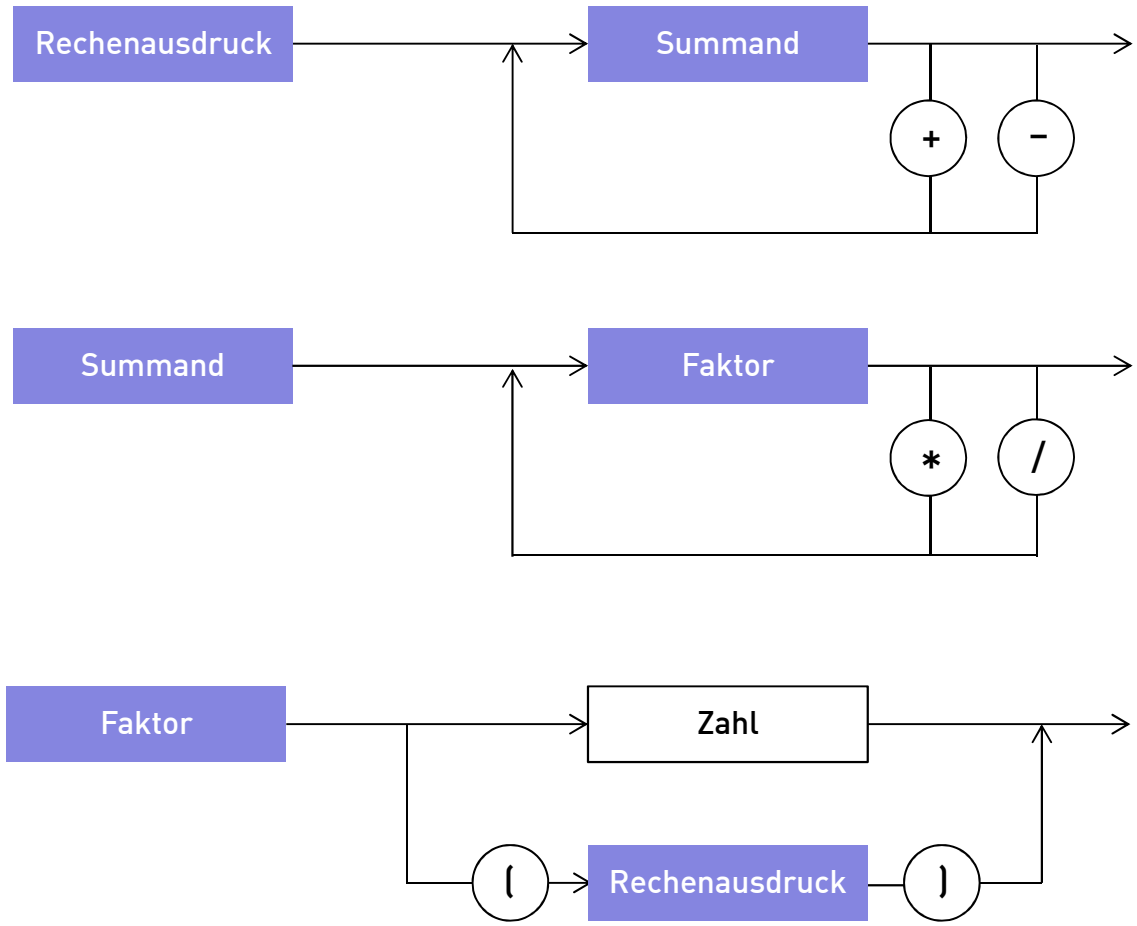
Die formale Grammatik der vier Grundrechenarten

Der Parserbaum für den Ausdruck `(12+3)*4`.



Die formale Grammatik der vier Grundrechenarten

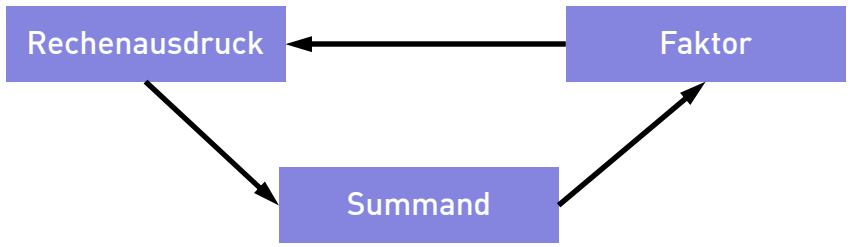
Eine anschauliche Darstellung.

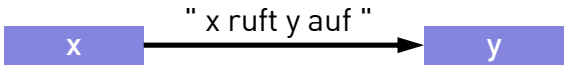


Die formale Grammatik der vier Grundrechenarten

Erkenntnisse.

- 7 mögliche Arten von Eingabesymbolen (*Token*)
  - Zahl
  - + - * / ( )
- 3 erforderliche Auswertungsfunktionen
  - Rechenausdruck | Summand | Faktor
- Sog. indirekter rekursiver Abstieg im Kontrollfluss





Eingabesymbole als Token

Als Token (Kategorie-Wert-Paar) können wir die eingegebenen Rechensymbole geeignet abspeichern.

- Die Zeichen der Eingabe werden zunächst einzeln eingelesen und dann zu sog. Token zusammen gesetzt.
  - Bemerkung: auch der C++ Compiler verarbeitet Ihren Quellcode zu Token.
- Z.B. soll aus den 13 Zeichen der Eingabe `1+4*(4.5-6/2)` die folgende Liste von Token erstellt werden:

Kategorie	Wert
zahl	1
plus	+
zahl	4
mal	*
linke klammer	(
zahl	4.5
minus	-
zahl	6
geteilt durch	/
zahl	2
rechte klammer	)

Token

Implementierung eines benutzerdefinierten Typs (d.h. einer Klasse) namens `Token` für Kategorie-Wert-Paare.

```
class Token                 // Benutzerdefinierter Typ namens Token.
{
  public:                   // Die folgenden Teile sind direkt zugreifbar.
    char kind;              // Art des Token, '9' soll fuer Zahl stehen.
    double value;           // Wert (nur fuer Zahl-Token).
};
```

- `Token` ist damit ein Typ (wie `int` oder `std::string`, oder wie `X_div` aus dem Beispiel in Teil 4 der Grundlagen) und kann verwendet werden, um Variablen zu definieren.
  - Das Schlüsselwort `class` bedeutet "benutzerdefinierter Typ" und zeigt an, dass jetzt ein Typ aus null oder mehr Teilen (sog. "Membern") definiert wird.
  - Die auf `public:` folgenden Teile des Typs sind "von überall" verwendbar (im Unterschied zu `private:`).
  - Token-Variable der obigen Definition bestehen aus zwei "von überall" verwendbaren *Datenmembern*. Das erste Member ist ein `char` namens `kind`, das zweite Member ist ein `double` namens `value`.

Token

Implementierung eines benutzerdefinierten Typs (d.h. einer Klasse) namens `Token` für Kategorie-Wert-Paare.

- Zur Erinnerung: die *Punktsyntax* für den Zugriff auf die (`public`) Member: `objekt_name.member_name`
- Quellcode-Beispiel

```
Token t1;
t1.kind = '+';
cout << t1.kind << endl;

Token t2;
t2.kind = '9'; t2.value = 43.21;
cout << t2.kind << ',' << t2.value << endl;

Token t3;
t3 = t2;
cout << t3.kind << ',' << t3.value << endl;
```

- Als `Token` werden wir z.B. den Rechenausdruck `(1.5+4)*11` so darstellen:

kind	'('	'9'	'+'	'9'	')'	'*'	'9'
value		1.5		4			11

  - Zahl-Token werden durch `kind=='9'` identifiziert (zugegeben etwas kryptisch).
  - Nur für Zahl-Token werden wir `value` verwenden.

Token

Verbesserung des `Token` Typs: Initialisierung durch Konstruktoren.

```
class Token {
  public:
     char kind;        // Die Kategorien der Eingabesymbole.
     double value;     // Bei Zahlenkategorie: der Wert.

     void print() const {
         kind=='9' ? cout << value : cout << kind; // Der ternaere Operator.
     }

     // Konstruktor, erstelle ein Token aus einem char:
     Token( char ch ) : kind{ch}, value{0.0} { /* mache nichts sonst */ }

     // Konstruktor, erstelle ein Zahl-Token aus einem double:
     Token( double val ) : kind{'9'}, value{val} { /* mache nichts sonst */ }

     // Standardkonstruktor, erstelle "das Standardtoken" ohne Vorgabe:
     Token( ) : Token{0.0} { /* mache nichts sonst */ }
};

// Anwendung:

Token t1{'+'}; t1.print(); cout << endl;

Token t2{43.21}; t2.print(); cout << endl;

Token t3{};
t3 = t1;
t3.print(); cout << endl;
```

Token

Verbesserung des `Token` Typs: Initialisierung durch Konstruktoren.

- Die `Token` Klasse hat drei Konstruktoren:

```
Token::Token( char ch ) : kind{ ch }, value{ 0.0 }
{}

Token::Token( double val ) : kind{ '9' }, value{ val }
{}

// Standardkonstruktor
Token::Token( ) : Token{0.0}
{}
```

- Konstruktoren sind besondere Memberfunktionen.
  - Zur *Initialisierung* von Objekten (hier `Token`-Objekten).
  - Jeder benutzerdefinierte Typ benötigt Konstruktoren.
  - Besonders den Standardkonstruktor.
- Konstruktoren tragen den selben Name wie die Klasse und dürfen keinen Rückgabewert haben.
- Die spezielle Syntax, die mit einem Doppelpunkt `:` beginnt (sog. *Member-Initialisierungsliste*), kann nur für Konstruktoren verwendet werden.

Tokenstrom

Ein Tokenstrom entsteht, indem eine Zeichenfolge (z.B. von `cin`) in die formal definierten Symbolen der Grammatik zerteilt wird.

```
class Token_stream {
   public:
     Token get( ); // Token aus cin erzeugen.
     void putback( Token t ) { // Token puffern.
         if( full ) error( "void Token_stream::putback(), Puffer voll" );
         buffer = t; full = true;
      }
   private:
     bool full;      // Ist ein Token im Puffer?
     Token buffer;   // Platz, um ein Token zu puffern.
};
```

- Wir definieren einen Tokenstrom als `class Token_stream`.
- Objekte vom Typ `Token_stream` lesen Eingaben vom `cin`–Strom ein, bestimmen daraus ein `Token` und geben dieses aus.
  - Beachten Sie, dass `Token_stream` in der Lage sein muss, ein `Token` auch zurück zu stellen, es wird also ein Tokenstrom-Puffer benötigt.
- *Gepufferte Tokenströme gehören allgemein zu den grundlegendsten Konzepten der Behandlung von Eingaben beim Programmieren.*

Tokenstrom

`Token Token_stream::get()` liest einzelne Zeichen ein, bestimmt daraus das nächste `Token` und gibt dieses zurück.

```
01 Token Token_stream::get( ) {
02     if( full ) { full = false; return buffer; }
03     char ch{ }; cin >> ch;
04     switch( ch ) {
05         case ';': // Ende eines Rechenausdrucks
06         case 'q': // Programmende
07         case '(': case ')': case '+': case '-': case '*': case '/':
08            return Token{ ch }; // jedes Zeichen ist sein eigenes Token
09         case '.': case '0': case '1': case '2': case '3': case '4':
10         case '5': case '6': case '7': case '8': case '9': // Zahl als Token
11         {
12            cin.putback( ch ); // der cin Strom kennt auch ein putback()
13            double val{ }; cin >> val;
14            return Token{ val };
15         }
16        default: error( "Token Token_stream::get(), unbekanntes Token" );
17    }
18 }
```

- Was passiert in der Zeile 2 in dem obigen Quellcode?
- Erklären Sie die Zeilen 12 bis 14 in dem obigen Quellcode.

Token und die formale Grammatik

Umsetzung.

- Die Token eines eingegebenen Rechenausdrucks können nun
  - mittels der `Token` Klasse einzeln erfasst werden,
    - ein geeigneter Typ für die einzelnen Rechensymbole (`Token`),
  - mittels der `Token_stream` Klasse eingelesen und erzeugt werden,
    - im Kern die Funktion, die aus dem gepufferten Zeicheneingabestrom `cin` der StdLib einen gepufferten Strom von `Token` generiert.
  - Probieren Sie es aus, z.B. so:

```
Token_stream ts{}; // Standardkonstruktor?!
Token tok{};
while( cin ) {
   tok = ts.get();
   tok.print();
}
```

- Was fehlt noch?
  - Die drei Funktionen zum Parsen des eingegebenen Rechenausdrucks (d.h. letztlich zur Berechnung des Ergebnisses).
- Auf Grundlage der vorgestellten formalen Grammatik für die Grundrechenarten kann dies nun programmiert werden.

Erweiterung der Grammatik für den Treiber

Für die einfachere Bedienung des `main()`-Treibers erweitern wir die Grammatik ein wenig.

Calculator :
    'q'
    Expression_list **'q'**

Expression_list :
    Expression **';'**
    Expression **';'** Expression_list

Expression :
    Expression **'+'** Term
    Expression **'-'** Term
    Term

Term :
    Term **'*'** Primary
    Term **'/'** Primary
    Primary

Primary :
    Number
    '(' Expression **')'**

Number :
    floating-point literal

Die Funktion für einen Rechenausdruck

Wird von `main()` und `faktor()` aufgerufen, ruft `summand()` auf.

```
double rechenausdruck( Token_stream& ts ) { // "plus" und "minus" behandeln
    double li{ summand( ts ) };
    Token t{ ts.get( ) };

    while( true ) {
        switch( t.kind ) {
            case '+':     li += summand( ts );
                          t = ts.get( );
                          break;
            case '-':     li -= summand( ts );
                          t = ts.get( );
                          break;
            default:      ts.putback( t );
                          return li;
        }
    }
}
```

- Gehen Sie gedanklich in Ruhe ("offline") durch die Quellcodes.

Die Funktion für einen Summand

Wird von `rechenausdruck()` aufgerufen, ruft `faktor()` auf.

```
double summand( Token_stream& ts ) { // "mal" und "geteilt durch" behandeln
    double li{ faktor( ts ) };
    Token t{ ts.get( ) };

    while( true ) {
        switch( t.kind ) {
            case '*': li *= faktor( ts );
                      t = ts.get( );
                      break;
            case '/':
            {
               double d{ faktor( ts ) };
               if( d==0 ) error( "Div. durch 0!" );
               li /= d;
               t = ts.get( );
               break;
            }
            default: ts.putback( t ); return li;
        }
    }
}
```

- Gehen Sie gedanklich in Ruhe ("offline") durch die Quellcodes.

Die Funktion für einen Faktor

Wird von `summand()` aufgerufen, ruft `rechenausdruck()` auf.

```
double faktor( Token_stream& ts ) { // Zahlen und Klammern behandeln
    Token t{ ts.get( ) };
    switch( t.kind ) {
        case '9': return t.value;
        case '(':
        {
          double d{ rechenausdruck( ts ) };
          t = ts.get( );
          if( t.kind != ')' ) error( ") erwartet!" );
          return d;
        }
        default: error( "Faktor erwartet!" );
    }
}
```

- Gehen Sie gedanklich in Ruhe ("offline") durch die Quellcodes.
- Warum werden hier nur die beiden Fälle "öffnende Klammer" und "Ziffer" unterschieden?

Der Treiber `main()`

Wird vom Benutzer / Betriebssystemebene aufgerufen, ruft `rechenausdruck()` auf.

```
// try ...

    Token_stream ts{}; // Standardkonstruktor?!
    Token tok{};

    while( true ) {
        tok = ts.get();
        if( tok.kind == 'q' ) break;
        ts.putback( tok );
        cout << '=';
        cout << rechenausdruck( ts ) << endl;
    }

    return 0;

// catch ...
```

Übung

Bringen Sie das Mini-Rechner Programm zur Ausführung.

- Finden Sie den kleinen, eingebauten Fehler und versuchen Sie, ihn zu beheben.
  - Hinweis: `;`

Einige Beispielfragen

Fallstudie: Taschenrechner.

- Nennen Sie die fünf grundsätzlichen Schritte der Softwareentwicklung und beschreiben Sie diese kurz.
- Erklären Sie das iterativ-inkrementelle Vorgehen bei der Entwicklung von Software.
- Warum ist es eine gute Idee, zunächst eine funktional stark eingeschränkte, aber voll lauffähige Version eines Programms zu entwickeln? Wie würden Sie eine solche Version nennen?
- Was ist eine Grammatik? Was ist ein Token? Wie hängen Token und formale Grammatiken zusammen?
- Was ist eine Klasse? Wozu verwendet man Klassen?
- Was ist ein Konstruktor?
- Was sind die Member einer Klasse?

Einige Beispielfragen

Fallstudie: Taschenrechner.

- Beschreiben Sie den Unterschied zwischen *Faktor*, *Rechenausdruck*, *Summand* und *Zahl* in Anlehnung an die durchgenommenen Inhalte.
- Warum hat das Mini-Rechner Programm keine Funktion `zahl()` für die Behandlung der Zahlen aus der Grammatik?
- Warum wird in der `rechenausdruck()` Funktion im `default` Teil der `switch` Anweisung das Token zurückgelegt? Wohin?
- Warum gibt es in `rechenausdruck()` und `summand()` jeweils eine `while( true )` Schleife, in `faktor()` aber nicht?
- Was passiert in der `Token_stream` Klasse, wenn ein Token im Puffer liegt und die Memberfunktion `get()` aufgerufen wird?

Einige Beispielfragen

Fallstudie: Taschenrechner.

- Warum teilt man Quellcode in mehrere Funktionen auf?
- Wie entscheiden Sie, was eine eigenständige Funktion ist?
- Warum empfiehlt es sich, ein Programm schrittweise zu erstellen?
- Wann fangen Sie an zu testen?
- Wann ist ein Programm fertig?
- Warum war es so schwierig, eine Lösung für das Mini-Rechner Programm zu finden?
- Fügen Sie in der besprochenen deutschen Teilgrammatik den bestimmten Artikel "die" ein, so dass Sätze wie "Die Feuerwehrmänner löschen und die Computer rechnen" gültig sind.
- Sie haben zwei der wichtigsten fundamentalen Prinzipien zum Umgang mit Eingaben beim Programmieren kennen gelernt, *gepufferte Tokenströme* und *Parsen / formale Grammatiken*. Erklären Sie die beiden Konzepte.

Nächste Einheit:
Funktionen und Programmstruktur