Lernziele:
Das primäre Ziel ist es, den Workflow "Bearbeiten (und speichern) -> Übersetzen -> Ausführen (sofern beim Übersetzen keine Fehler aufgetreten sind)" zu üben. Programme sollen durch die Methode "kleines Beispiel wird schrittweise erweitert" umgesetzt werden.

Folgende sekundäre Ziele werden ebenfalls berücksichtigt:
- Üben der Inhalte der Vorlesung (wie in den Videos gezeigt)
- Erlernen neuer Spracheigenschaften anhand von Beispielen

Schritt 1

In **<stdio.h>** sind auch die Funktionen **getchar** und **putchar** deklariert:
- Die Funktion **getchar** liest ein Zeichen von der Tastatur ein und gibt es als Rückgabewert zurück.
- Die Funktion **putchar** erwartet ein Zeichen als Argument und gibt es auf dem Bildschirm aus.

Das rechts abgebildete Programm **ex1.abc** macht in einer Endlosschleife folgendes: Ein Zeichen einlesen und dann ausgeben, ein Zeichen einlesen und dann ausgeben, usw.

```
@ <stdio.hdr>

fn main()
{
    while (true) {
        putchar(getchar());
    }
}
```

Aufgaben:
1. Schreibe das Programm in deinem bevorzugten Editor. Übersetze und führe es aus. Beachte dabei bewusst folgendes: Zeichen werden offensichtlich nicht sofort ausgegeben, sondern erst nachdem du Return gedrückt hast. Wenn du zum Beispiel 'a', 'b', 'c' und dann Return eingibst, erhältst du folgende Ausgabe:

   ```
   MCL:session1 lehn$ ./a.out
   abc
   abc
   ```

2. Mit "Control-C" kannst du das Programm (gewaltsam) abbrechen, da es sich um eine Endlosschleife handelt.
3. Unsere Programme laufen innerhalb der Shell. Deshalb waren die obigen Aussagen über **getchar** und **putchar** falsch. Richtiger (im Sinne der Fuzzy-Logik) ist:
   (A) **getchar** bekommt die Eingabe nicht von der Tastatur, sondern von der Shell.
   (B) **putchar** schreibt Zeichen nicht auf den Bildschirm, sondern gibt sie für die Ausgabe an die Shell weiter.
   (C) Standardmäßig gibt die Shell die Eingabe erst an unser Programm weiter, wenn du Return eingibst. Der Zweck ist, dass du vor dem Drücken von Return noch die Möglichkeit hast, deine Eingabe (zum Beispiel mit Backspace) zu korrigieren.
   (D) Mit dem Shell-Befehl **stty** kannst du dieses Verhalten so ändern, dass die Eingabe sofort an das Programm weitergeleitet wird:

   ```
   MCL:session1 lehn$  stty -icanon min 1
   MCL:session1 lehn$  ./a.out
   aabbcc
   ```

   (E) Die Shell ist jetzt "unbrauchbar" (wir wollen natürlich unsere Eingaben vor dem Drücken von Return korrigieren können). Deshalb solltest du das Terminal schließen und neu starten.
4. Im Programm wurde das Schlüsselwort true als Bedingung verwendet. Vom Compiler wird intern true mit 1 ersetzt. Allgemein kann man als Bedingung einen beliebigen Integer-Wert verwenden. Dabei gilt "Null ist false und alles andere ist true". Das testen wir über den Workflow "Bearbeiten und speichern -> Übersetzen -> Testen":
   (A) Ändere **true** in **42**.
   (B) Ändere **true** in **2 > 1**.

Schritt 2

Zeichen sind in Wirklichkeit nur Zahlenwerte, die als Zeichen interpretiert werden. Zum Beispiel hat 'A' den Wert 65 und 'B' den Wert 66. Diese Codierung kann man in einer ASCII-Tabelle nachschlagen. Alternativ kann man auch ein Programm schreiben, um den Zahlenwert auszugeben!

Aufgaben:

1. Ändert das vorhandene Programm wie folgt:

```
@ <stdio.hdr>

fn main()
{
    while (true) {
        local ch: int = getchar();
        printf("ch = %d\n", ch);
    }
}
```

2. Übersetze das Programm und testet es. Wenn du jetzt 'a', 'b', 'c' und dann Return eingibst, erhältst du:

```
MCL:session1 lehn$  ./a.out
abc
ch = 97
ch = 98
ch = 99
ch = 10
```

   Der ASCII-Wert von 'a' ist also 97, der von 'b' ist 98, der von 'c' ist 99 und der von Return ist 10. Bestimme die ASCII-Werte der Ziffern '0', '1', …, '9'.

3. Es ist unschön, dass unser Programm jedes Mal mit Control-C abgebrochen werden muss, um es zu beenden. Ändere das Programm wie folgt:

```
@ <stdio.hdr>

fn main()
{
    while (true) {
        local ch: int = getchar();
        if (ch == '.') {
            break;
        } else {
            printf("ch = %d\n", ch);
        }
    }
}
```

   Teste es mit einer Eingabe, die ein '.' enthält, zum Beispiel mit 'a', 'b', 'c', '.'. Nun bricht das Programm ab, wenn das Zeichen '.' gelesen wurde.

4. Es wäre noch besser, wenn das Programm abbricht, wenn keine weitere Eingabe vom Benutzer erwartet wird. Ändere im Programm die if-Bedingung wie folgt:

```
if (ch == EOF) {
```

   Das Programm kannst du jetzt mit der Eingabe von Control-D beenden. Das scheint nicht besser als Control-C zu sein, aber warte mal ab.

5. **EOF** steht für "End Of File". Probiere Folgendes aus (wenn die Datei mit deinem ABC-Programm nicht ex1.abc heißt, ändere das entsprechend ab):

```
MCL:session1 lehn$ ./a.out < ex1.abc
ch = 64
ch = 32
…
```

6. Vom Compiler wird EOF mit einem Zahlenwert ersetzt. Der Wert ist plattformabhängig, aber in der Regel ist es der Wert -1. Teste, ob sich das Programm noch gleich verhält, wenn du -1 statt EOF schreibst. Verwende aber anschließend wieder EOF statt -1. Das ist lesbarer.

7. Ändere den Format-String und die printf-Anweisung wie folgt:

```
@ <stdio.hdr>

fn main()
{
    while (true) {
        local ch: int = getchar();
        if (ch == EOF) {
            break;
        } else {
            printf("ch = %d -> '%c'\n", ch, ch);
        }
    }
}
```

   Jetzt wird einmal der ASCII-Wert (beim Platzhalter %d) und einmal das zugehörige Zeichen (beim Platzhalter %c) ausgegeben.

8. Ändere das Programm wie folgt:

```
@ <stdio.hdr>

fn main()
{
    while (true) {
        local ch: int = getchar();
        if (ch >= '0' && ch <= '9') {
            printf("digit '%c'\n", ch);
        } else if (ch == EOF) {
            break;
        } else {
            printf("ch = %d -> '%c'\n", ch, ch);
        }
    }
}
```

   Damit werden bei der Eingabe die Ziffern '0' bis '9' speziell behandelt. Teste das Programm zum Beispiel mit einer Eingabe wie "a1cd4". Dann solltest du folgende Ausgabe bekommen:

```
MCL:session1 lehn$ ./a.out
a1cd4
ch = 97 -> 'a'
digit '1'
ch = 99 -> 'c'
ch = 100 -> 'd'
digit '4'
ch = 10 -> '
'
```

9. Neben dem ASCII-Wert einer Ziffer soll jetzt auch der Zahlenwert der Ziffer ausgegeben werden. Mit "Zahlenwert der Ziffer ausgeben" ist folgendes gemeint:
   - Das Zeichen '0' hat den ASCII-Wert 48, es soll aber 0 ausgegeben werden
   - Das Zeichen '1' hat den ASCII-Wert 49, es soll aber 1 ausgegeben werden
   - …
   - Das Zeichen '9' hat den ASCII-Wert 57, es soll aber 9 ausgegeben werden

   Das kann man erreichen, indem man bei jeder Ziffer den ASCII-Wert von '0' abzieht. Ändere die Zeile mit der printf-Anweisung wie folgt:

```
printf("digit '%c', value = %d\n", ch, ch - '0');
```

   Teste das Programm wieder mit einer Eingabe wie "a1cd4".

Schritt 3

Jetzt wird es etwas komplexer. Statt einzelner Ziffern sollen nun ganze Zahlenwerte in der Eingabe erkannt werden. Ihr bekommt dafür eine Vorlage. Diese sollt ihr erst testen, die Details schauen wir uns dann anschließend etwas genauer an.

Ändert bzw. ergänzt euer Programm wie folgt:

```
@ <stdio.hdr>

fn main()
{
    while (true) {
        local ch: int = getchar();
        if (ch >= '0' && ch <= '9') {
            local val: int = 0;
            while (ch >= '0' && ch <= '9') {
                val = val * 10;
                val = val + ch - '0';
                ch = getchar();
            }
            printf("integer with value %d\n", val);
        } else if (ch == EOF) {
            break;
        } else {
            printf("ch = %d -> '%c'\n", ch, ch);
        }
    }
}
```

Aufgaben:

1. Testet das Programm mit einer Eingabe wie zum Beispiel "abc123xy42". Dann sollte korrekterweise erkannt werden, dass die Zahlenwerte 123 und 42 vorkommen:

   ```
   MCL:session1 lehn$ ./a.out
   ab123xy42
   ch = 97 -> 'a'
   ch = 98 -> 'b'
   integer with value 123
   ch = 121 -> 'y'
   integer with value 42
   ```

   Aber das Programm arbeitet noch nicht korrekt. Das 'x' wurde "verschluckt". Allgemein wird nach jeder Zahl das nachfolgende Zeichen verschluckt. Darum kümmern wir uns später.
2. Betrachtet nun den Programmteil, der Zahlen in der Eingabe entdeckt und dafür sorgt, dass die Variable **val** anschließend den zugehörigen Zahlenwert enthält. Betrachtet dazu diesen Teil:

   ```
   local val: int = 0;
   while (ch >= '0' && ch <= '9') {
       val = val * 10;
       val = val + ch - '0';
       ch = getchar();
   }
   ```

   Erklärt eurem Nebensitzer (oder lasst es euch von uns erklären), wieso bei einer der Eingabe der Ziffernfolge '1', '2', '3' die Variable **val**:
   (A) nach dem ersten Schleifendurchlauf den Wert 1 hat,
   (B) nach dem zweiten Schleifendurchlauf den Wert 12,
   (C) nach dem zweiten Schleifendurchlauf den Wert 123.
3. Verwendet in der While-Schleife die Operatoren *= und +=
4. Erklärt nun eurem Nebensitzer, wieso nach jeder Zahl ein Zeichen verschluckt wird. Wie das Problem gelöst werden kann, seht ihr im nächsten Schritt.

Schritt 4
Das Problem mit verschluckten Zeichen nach einer Ziffernfolge wurde in folgender Anpassung behoben:

```
 1 @ <stdio.hdr>
 2
 3 fn main()
 4 {
 5     local ch: int = getchar();
 6     while (true) {
 7         if (ch >= '0' && ch <= '9') {
 8             local val: int = 0;
 9             while (ch >= '0' && ch <= '9') {
10                 val *= 10;
11                 val += ch - '0';
12                 ch = getchar();
13             }
14             printf("integer with value %d\n", val);
15         } else if (ch == EOF) {
16             break;
17         } else {
18             printf("ch = %d -> '%c'\n", ch, ch);
19             ch = getchar();
20         }
21     }
22 }
```

Aufgaben:
1. Testet, dass jetzt tatsächlich kein Zeichen mehr "verschluckt" wird.
2. In dem Programm gibt es zwei wesentliche Änderungen:
   (A) Die Variable **ch** wird jetzt vor der While-Schleife deklariert. Der Scope der Variable **ch** ist nun die gesamte Funktion. Stellen, an denen **ch** überschrieben wird, sind in den Zeilen 12, 19.
   (B) Im Else-Zweig der If-Anweisung wird nun (Zeile 19) ebenfalls **getchar** aufgerufen und **ch** überschrieben. Testet, wieso das notwendig ist, indem ihr diese Zeile auskommentiert, d.h. abändert in

   ```
   // ch = getchar();
   ```

   Dann testet das Programm erneut. Erklärt dem Nebensitzer, was schief geht und wieso. Macht die Änderung anschließend wieder rückgängig.
3. Das Programm soll jetzt auch das Zeichen '+' speziell behandeln. Fügt dazu nach Zeile 14 den folgenden Code ein:

   ```
   } else if (ch == '+') {
       printf("PLUS\n");
       ch = getchar();
   ```

   Testet das Programm zum Beispiel mit "12 + 4".
4. Ändert das Programm so ab, dass:
   (A) ein '*' die Ausgabe ASTERISK ("Sternchen") erzeugt.
   (B) ein '(' die Ausgabe LPAREN (für "Left Parenthesis" also "Linke Klammer") erzeugt.
   (C) ein ')' die Ausgabe RPAREN (für "Right Parenthesis" also "Rechte Klammer") erzeugt.
   (D) ein ';' die Ausgabe SEMICOLON erzeugt.

Schritt 5
Jetzt kommt eine größere Änderung:

1. Der Code innerhalb der While-Schleife wird in eine Funktion **getToken** ausgelagert.
2. Alle Variablen sind jetzt global. Und es gibt eine weitere Variable **token**. Diese Variable wird jedes Mal überschrieben, wenn in der Eingabe etwas Interessantes entdeckt wurde, zum Beispiel auf 1 bei einem Integer, auf 2 bei einem Plus, usw.
3. In **main** wird in einer Schleife **getToken** aufgerufen und der Wert von **token** ausgegeben. Die Schleife wird verlassen, wenn **token** den Wert 7 hat, da dann getToken ein **EOF** gefunden hat.

```
1  @ <stdio.hdr>
2
3  global token: int = 0;
4  global ch: int = 0;
5  global val: int = 0;
6
7  fn getToken()
8  {
9      if (ch >= '0' && ch <= '9') {
10         val = 0;
11         while (ch >= '0' && ch <= '9') {
12             val = val * 10;
13             val = val + ch - '0';
14             ch = getchar();
15         }
16         printf("integer with value %d\n", val);
17         token = 1;
18     } else if (ch == '+') {
19         printf("PLUS\n");
20         ch = getchar();
21         token = 2;
22     } else if (ch == '*') {
23         printf("ASTERISK\n");
24         ch = getchar();
25         token = 3;
26     } else if (ch == ';') {
27         printf("SEMICOLON\n");
28         ch = getchar();
29         token = 4;
30     } else if (ch == '(') {
31         printf("LPAREN\n");
32         ch = getchar();
33         token = 5;
34     } else if (ch == ')') {
35         printf("RPAREN\n");
36         ch = getchar();
37         token = 6;
38     } else if (ch == EOF) {
39         token = 7;
40     } else {
41         printf("ch = %d -> '%c'\n", ch, ch);
42         ch = getchar();
43         token = 8;
44     }
45 }
46
47 fn main()
48 {
49     while (true) {
50         getToken();
51         if (token == 7) {
52             break;
53         } else if (token == 1) {
54             printf("token = %d, val = %d\n", token, val);
55         } else {
56             printf("token = %d\n", token);
57         }
58     }
59 }
```

Aufgaben:

1. Über den Sinn und Unsinn dieser Änderung machen wir uns zunächst keine Gedanken. Zunächst trainieren wir unsere Copy-Und-Paste-Skills und das Bedienen unseres Editors.
2. Fügt nach Zeile 52 folgenden Code ein:

```
53         } else if (token == 1) {
54             printf("token = %d, val = %d\n", token, val);
```

3. Entfernt in **getToken** alle Zeilen mit einem **printf**.

Schritt 6

Mit einer sogenannten Enum-Deklaration kann man Symbole für Zahlenwerte vereinbaren. Der Compiler ersetzt dann diese Symbole mit dem Zahlenwert. Ein Beispiel kennen wir schon. **EOF** wird vom Compiler mit -1 ersetzt. Der Sinn von solchen Konstanten ist, dass der Code damit lesbarer wird.

In unserem Beispiel könnte folgende Enum-Deklaration nützlich sein:

```
enum TokenKind {
    EOI, // = 0 (end of input)
    BAD,
    INTEGER,
    PLUS,
    ASTERISK,
    SEMICOLON,
    LPAREN,
    RPAREN,
};
```

Das Symbol **EOI** ("End of Input") wird dann mit 0, das Symbol **BAD** mit 1, **INTEGER** mit 2, usw. ersetzt. Die Symbole werden also vom Compiler beginnend mit Null fortlaufend durchnummeriert.

Aufgaben:

1. Code anpassen und testen.
2. Falls keine Zahl, '+', '*', '(' oder ')' gefunden wird, setzt **getToken** die Variable token auf **BAD**. Statt dessen soll jetzt ein "ungültiges" Zeichen aber ignoriert werden und das nächste "interessante" gesucht werden. Ersetzt dazu Zeile 47 (in der token mit **BAD** überschrieben wird) mit:

   ```
   47          getToken();
   ```

   Insbesondere werden jetzt also Leerzeichen und Zeilenumbrüche ignoriert, wie folgender Test zeigt:

   ```
   MCL:session1 lehn$ ./a.out
   12 + 3 * (4 + 5);
   token = 2, val = 12
   token = 3
   token = 2, val = 3
   token = 4
   token = 6
   token = 2, val = 4
   token = 3
   token = 2, val = 5
   token = 7
   token = 5
   ```

```
1  @ <stdio.hdr>
2
3  enum TokenKind {
4      EOI, // = 0 (end of input)
5      BAD,
6      INTEGER,
7      PLUS,
8      ASTERISK,
9      SEMICOLON,
10     LPAREN,
11     RPAREN,
12 };
13
14 global token: int = 0;
15 global ch: int = 0;
16 global val: int = 0;
17
18 fn getToken()
19 {
20     if (ch >= '0' && ch <= '9') {
21         val = 0;
22         while (ch >= '0' && ch <= '9') {
23             val = val * 10;
24             val = val + ch - '0';
25             ch = getchar();
26         }
27         token = INTEGER;
28     } else if (ch == '+') {
29         ch = getchar();
30         token = PLUS;
31     } else if (ch == '*') {
32         ch = getchar();
33         token = ASTERISK;
34     } else if (ch == ';') {
35         ch = getchar();
36         token = SEMICOLON;
37     } else if (ch == '(') {
38         ch = getchar();
39         token = LPAREN;
40     } else if (ch == ')') {
41         ch = getchar();
42         token = RPAREN;
43     } else if (ch == EOF) {
44         token = EOI;
45     } else {
46         ch = getchar();
47         token = BAD;
48     }
49 }
50
51 fn main()
52 {
53     while (true) {
54         getToken();
55         if (token == EOI) {
56             break;
57         } else if (token == INTEGER) {
58             printf("token = %d, val = %d\n", token, val);
59         } else {
60             printf("token = %d\n", token);
61         }
62     }
63 }
```

Ausblick

Was wir in **getToken** programmiert haben, nennt man einen Lexer. Dieser gruppiert einzelne Zeichen in sogenannte Tokens. Bei der Implementierung haben wir einen sogenannten endlichen Automaten umgesetzt.

Der nächste Schritt könnte nun sein, den Lexer zu verwenden, um einen Taschenrechner zu implementieren, der arithmetische Ausdrücke (mit +, * und Klammern) auswerten kann.

Alternativ dazu könnte der Taschenrechner Assembler-Code erzeugen, der beschreibt, wie man den Ausdruck auswertet. Wenn man den rechts abgebildeten Code nach der Funktion **getToken** in unser Programm einfügt, wird genau das erreicht:

```
MCL:session1 lehn$ ./a.out
3 * (4 + 5);
        load 3, %1
        load 4, %2
        load 5, %3
        add %3, %2, %4
        mul %4, %1, %5
```

So ein Programm könnte man schon einen Compiler nennen. Ein guter Compiler würde in dem Fall natürlich den Ausdruck tatsächlich ausrechnen und Code wie diesen erzeugen:

```
MCL:session1 lehn$ ./a.out
3 * (4 + 5);
        load 27, %1
```

```
51 global reg: int;
52
53 fn parseSum(): int;
54
55 fn parseInt(): int
56 {
57     if (token == INTEGER) {
58         getToken();
59         printf("\tload %d, %%%d\n", val, ++reg);
60         return reg;
61     } if (token == LPAREN) {
62         getToken();
63         local sum: int = parseSum();
64         if (sum < 0) {
65             printf("sum expected\n");
66             return -1;
67         }
68         if (token != RPAREN) {
69             printf("expected ')'\n");
70             return -1;
71         }
72         getToken();
73         return sum;
74     } else {
75         return -1;
76     }
77 }
78
79 fn parseProd(): int
80 {
81     local prod: int = parseInt();
82     if (prod < 0) {
83         return -1;
84     }
85     while (token == ASTERISK) {
86         getToken();
87         local factor: int = parseInt();
88         if (factor < 0) {
89             printf("expected factor\n");
90             return -1;
91         }
92         printf("\tmul %%%d, %%%d, %%%d\n", factor, prod, ++reg);
93     }
94     return reg;
95 }
96
97 fn parseSum(): int
98 {
99     local sum: int = parseProd();
100    if (sum < 0) {
101        return -1;
102    }
103    while (token == PLUS) {
104        getToken();
105        local summand: int = parseProd();
106        if (summand < 0) {
107            printf("expected summand\n");
108            return -1;
109        }
110        printf("\tadd %%%d, %%%d, %%%d\n", summand, sum, ++reg);
111    }
112    return reg;
113 }
114
115 fn main()
116 {
117     getToken();
118     for (local sum: int = -1; (sum = parseSum()) >= 0; ) {
119         if (token == SEMICOLON) {
120             getToken();
121         }
122     }
123 }
```