Lernziele:
- Verständnis für die lexikalische Analyse (Umwandlung von Zeichenketten in Tokens)
- Vertiefung der Programmierkenntnisse

Vorbereitung: Compiler aktualisieren

1. Wechselt in das Verzeichnis **abc-llvm** mit dem Source-Code für den Compiler.
2. Führt das Kommando **git pull** aus.
3. Führt das Kommando **make** aus.
4. Führt das Kommando **sudo make install** aus.

Ausgangspunkt

Als Ausgangspunkt verwenden wir den in der Vorlesung vorgestellten Code (**lexer.hdr**, **lexer.abc**, **xtest_lexer.abc**) für den Lexer. Hier wurde der Code in **lexer.abc** leicht umstrukturiert in einen internen und einen extern zugänglichen Teil:

lexer.hdr

```
enum TokenKind
{
    BAD_TOKEN,
    EOI,
    DECIMAL_LITERAL,
    PLUS,
};

extern token: TokenKind;

extern fn getToken(): TokenKind;
```

lexer.abc

```
@ <stdio.hdr>
@ "lexer.hdr"

// -- internal: only available within this translation unit

global ch: int;

// -- exported: available for other translation units

global token: TokenKind;

fn getToken(): TokenKind
{
    while (ch == 0 || ch == ' ' || ch == '\n' || ch == '\t') {
        ch = getchar();
    }
    if (ch >= '0' && ch <= '9') {
        while (ch >= '0' && ch <= '9') {
            ch = getchar();
        }
        return token = DECIMAL_LITERAL;
    } else if (ch == '+') {
        ch = getchar();
        return token = PLUS;
    } else if (ch == EOF) {
        return token = EOI;
    } else {
        ch = getchar();
        return token = BAD_TOKEN;
    }
}
```

Im Terminal kann man das Testprogramm **xtest_lexer** erzeugen mit

abc xtest_lexer.abc lexer.abc

Da wir weitere Testprogramme schreiben (und noch keine Makefiles verwenden), empfiehlt es sich, das Testprogramm wie folgt zu übersetzen (von der Shell wird **[^x]*.abc** durch alle Dateien im aktuellen Verzeichnis ersetzt, die nicht mit "x" beginnen und die Dateierweiterung "abc" haben):

abc xtest_lexer.abc [^x]*.abc

Schritt 1: Vereinfachung des Testprogramms und Switch-Statements

Das Testprogramm soll so vereinfacht werden, dass nicht mehr durch eine Vielzahl von If-Anweisungen der Wert "token" auf einen String abgebildet werden muss. Stattdessen soll dafür eine Funktion "tokenKindStr" wie folgt verwendet werden:

```
@ <stdio.hdr>
@ "lexer.hdr"

fn main()
{
    while (getToken() != EOI) {
        printf("%s\n", tokenKindStr(token));
    }
}
```

Die Funktion "tokenKindStr" soll in "lexer.abc" implementiert werden. Statt der bisher verwendeten If-Anweisungen soll an dieser Stelle das Switch-Statement eingeführt werden. Auf die Details gehen wir später ein, zunächst könnt ihr diesen Code direkt übernehmen und in "lexer.abc" am Ende einfügen:

```
fn tokenKindStr(token: TokenKind): -> char
{
    local str: -> char = "??";
    switch (token) {
        case BAD_TOKEN:
            str = "BAD_TOKEN";
            break;
        case EOI:
            str = "EOI";
            break;
        case DECIMAL_LITERAL:
            str = "DECIMAL_LITERAL";
            break;
        case PLUS:
            str = "PLUS";
            break;
    }
    return str;
}
```

Aufgaben:

1. Passt den Header an, so dass diese Funktion extern verwendet werden kann.
2. **Wiederholung von Altbekanntem:**
   A. Macht euch klar, dass der Parameter "token" und die globale Variable "token" unterschiedliche Variablen sind (siehe "Scope einer Variable").
   B. "str" ist ein Zeiger auf ein Zeichen, "BAD_TOKEN" ist ein Array von Zeichen (dieses Array wird vom Compiler als globales Array definiert). Bei der Zuweisung str = "BAD_TOKEN" findet also ein impliziter Cast statt; "str" wird die Adresse des ersten Zeichens zugewiesen.
3. Bei einem Switch-Statement kann abhängig vom Wert eines Ausdrucks (hier ist der Ausdruck "token") zu einem case-Label innerhalb des Switch-Blocks gesprungen werden. Der Typ des Ausdrucks muss ganzzahlig sein (keine Fließkommazahlen, keine Structs, etc.) und der bei den case-Labels verwendete Wert muss konstant sein. Wird ein case-Label angesprungen, wird dort der Kontrollfluss fortgesetzt, bis entweder das Ende des Switch-Blocks erreicht wird oder ein break-Statement ausgeführt wird, mit dem der Switch-Block verlassen wird.
   1. **Fall-Through in Switch-Statements:** Entfernt die break-Statements innerhalb des Switch-Blocks. Testet das Programm mit verschiedenen Eingaben.
   2. **Wertebereich von Enums:** Fügt in **xtest_lexer.abc** in der Funktion **main** folgende Anweisung (vor der while-Schleife) ein:
      ```
      printf("%s\n", tokenKindStr(42));
      ```
      Für den konstanten Wert 42 wurde in TokenKind kein Symbol vereinbart. Dennoch können Variablen vom Typ TokenKind diesen Wert enthalten!
   3. Entfernt ein (beliebiges) case-Label. Der Compiler sollte sich jetzt beschweren, dass nicht alle für **TokenKind** deklarierten Konstanten in den case-Labels abgedeckt sind.

Schritt 2: Default-Label in Switch-Statements

Ändere die Funktion **tokenKindStr** wie folgt:

```
fn tokenKindStr(token: TokenKind): -> char
{
    switch (token) {
        case BAD_TOKEN: return "BAD_TOKEN";
        case EOI: return "EOI";
        case DECIMAL_LITERAL: return "DECIMAL_LITERAL";
        case PLUS: return "PLUS";
        default: return "??";
    }
}
```

Im Switch-Block wird jetzt ein default-Label verwendet, das immer dann angesprungen wird, wenn keiner der case-Labels zutrifft. Die Funktion hat nun ebenfalls mehrere Exit-Points, da nach jedem Label die Funktion mit einem Return-Statement verlassen wird. Mehrere Exit-Points können den Nachteil haben, dass es schwerer ist, den Kontrollfluss nachzuvollziehen. Im Gegenzug hat man in diesem Fall den Vorteil, dass keine unbeabsichtigten Fall-Throughs auftreten können.

Aufgaben

1. Bestätige, dass es nicht notwendig ist, das default-Label als letztes aufzuführen. Verschiebe die entsprechende Zeile zum Beispiel an den Anfang des Switch-Blocks.
2. Entferne eine Zeile mit einem case-Label. Bestätige, dass der Compiler sich nun nicht mehr beschwert, dass nicht alle Enum-Konstanten mit den case-Labels abgedeckt sind.
3. Verwende für die Implementierung von tokenKindStr deinen bevorzugten Stil.

Schritt 3: Ein neues Token

Der Lexer soll nun das Zeichen '*' als Token erkennen. Als Enum-Konstante soll dafür **ASTERISK** verwendet werden. Mache alle dafür notwendigen Anpassungen (damit ist eingeschlossen, dass es möglich ist zu testen, ob das Token erkannt wird).

Schritt 4: Zeilen und Spalten-Position eines Tokens (Vorbereitung)

Für ein gefundenes Token soll der Lexer nun ebenfalls festhalten, in welcher Zeile und Spalte dieses gefunden wurde. Dazu passen wir erst das Interface an und im nächsten Schritt die Implementierung.

In **lexer.hdr** wird ein Struct-Typ Token deklariert mit einem Member kind für die Art des Tokens und einem Membern für die Position. Der Typ der globalen Variable token wird auf **Token** geändert:

```
struct Token
{
    kind:   TokenKind;
    pos:    struct Pos {
                row, col: unsigned;
            };
};

extern token: Token;
```

In unserem Projekt müssen jetzt mehrere Stellen angepasst werden. Macht dazu zunächst nur folgende Änderungen:

1. In **xtest_lexer.abc** sollte das neue Feature getestet werden. Ersetzt die printf-Anweisung mit (der Zeilenumbruch war bei mir nötig, da mein Gehirn beim Programmieren nur Zeilen mit maximal 80 Zeichen verarbeiten kann):

```
printf("%u.%u: %s\n", token.pos.row, token.pos.col,
       tokenKindStr(token.kind));
```

2. In **lexer.abc** sollte bei der Definition von **token** der Typ auf **Token** geändert werden.
3. In **lexer.abc** sollte jedes Return-Statement so angepasst werden, dass **token.kind** gesetzt wird. Wurde zum Beispiel ein '+' gefunden, ändert sich das Return-Statement zu:

```
return token.kind = DECIMAL_LITERAL;
```

Dieser Schritt ist erfolgreich, wenn die Eingabe "123 +" zu folgender Ausgabe führt:

```
123 +
0.0: DECIMAL_LITERAL
0.0: PLUS
```

Schritt 5: Zeilen- und Spalten-Position eines Tokens (Implementierung)

Da wir jetzt das neue Feature testen können, macht es Sinn, mit der Implementierung zu beginnen. Die Idee ist, dass anstatt **getchar** eine eigene Funktion **nextCh** aufgerufen wird, die zwei Dinge macht:

1. Ein Zeichen mit **getchar** einlesen und damit **ch** überschreiben.
2. Die aktuelle Position in einer globalen Variabel **currentPos** aktualisieren.

Hier ein Blick in **lexer.abc** auf eine erste Implementierung von **nextCh** (Tabs werden dabei noch nicht richtig behandelt):

```
// -- internal: only available within this translation unit

global ch: int;
global currentPos: Pos = {1, 0};

fn nextCh()
{
    ch = getchar();
    if (ch == '\n') {
        ++currentPos.row;
        currentPos.col = 0;
    } else {
        ++currentPos.col;
    }
}
```

In **getToken** kann diese nun verwendet werden, indem man folgende Anpassungen macht:

- Nachdem alle Whitespaces ignoriert werden, setzt man **token.pos** auf **currentPos**:

```
while (ch == 0 || ch == ' ' || ch == '\n' || ch == '\t') {
    nextCh();
}
token.pos = currentPos;
```

- Alle Anweisungen der Form

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

ersetzt man mit

```
nextCh();
```

Setzt die Implementierung um und testet diese. Hier die Ausgabe eines möglichen Tests:

```
MCL:hpc0-abc lehn$ ./a.out
123 + *
1.1: DECIMAL_LITERAL
1.5: PLUS
1.7: ASTERISK
1 2 3
2.1: DECIMAL_LITERAL
2.3: DECIMAL_LITERAL
2.5: DECIMAL_LITERAL
```

Schritt 6: Festhalten der Zeichen, die zu einem Token gehören (Vorbereitung)

Aktuell kann unser Lexer zum Beispiel erkennen, dass Text eine Dezimalzahl enthält. In der Variable token werden aber nicht die Zeichen dieser Dezimalzahl festgehalten. Das soll nun geändert werden.

In "lexer.hdr" vereinbaren wir den Typ "TokenVal" für einen String mit 11 Zeichen (um die Unterstützung von Strings beliebiger Länge kümmern wir uns später) und ergänzen "Toke" um ein Member val von diesem Typen:

```
type TokenVal: array[12] of char;

struct Token
{
    kind:   TokenKind;
    pos:    struct Pos {
                row, col: unsigned;
            };
    val:    TokenVal;
};
```

In "xtest_lexer.abc" ersetzen wir die printf-Anweisung mit

```
        printf("%u.%u: %s (%s) \n", token.pos.row, token.pos.col,
               tokenKindStr(token.kind), token.val);
```

Damit sollte bereits alles wieder ohne Fehler übersetzen und sich das Programm auch ausführen lassen. Bestätige, dass token.val dabei immer ein leerer String ist und erkläre, ob die definiertes oder undefiniertes Verhalten ist.

Schritt 7: Festhalten der Zeichen, die zu einem Token gehören (Implementierung)

Ein erster Hack, um das Feature zu unterstützen, zeigt folgender Code-Ausschnitt von **lexer.abc**:

```
// -- internal: only available within this translation unit

global ch: int;
global currentPos: Pos = {1, 0};
global updateVal: bool;
global lengthVal: size_t;

fn nextCh()
{
    if (updateVal) {
        token.val[lengthVal++] = ch;
        token.val[lengthVal] = 0;
    }
    ch = getchar();
    if (ch == '\n') {
        ++currentPos.row;
        currentPos.col = 0;
    } else {
        ++currentPos.col;
    }
}
```

Es gibt zwei neue globale Variablen:

- Mit updateVal kann festgelegt werden, ob das aktuelle Zeichen zum Wert des Tokens hinzugefügt werden soll.
- Mit lengthVal wird festgehalten, wie viele Zeichen bereits zu token.val hinzugefügt wurden. Der Typ size_t ist dabei ein Unsigned-Integer-Typ und wird vom Compiler plattformabhängig festgelegt. Es wird garantiert, dass damit die Größe von jeder Variablen (also insbesondere die Größe eines Arrays) dargestellt werden kann.

In getToken kann man nun folgende Änderung vornehmen:

```
updateVal = false;
lengthVal = 0;
while (ch == 0 || ch == ' ' || ch == '\n' || ch == '\t') {
    nextCh();
}
token.pos = currentPos;
updateVal = true;
```

Beim Betreten der Funktion werden lengthVal und updateVal jeweils zurückgesetzt, dann werden alle Whitespaces ignoriert und danach alle weiteren Zeichen zu token.val hinzugefügt.

Aufgabe:

1. Im Code wird nicht überprüft, ob weitere Zeichen überhaupt noch zu token.val hinzugefügt werden können! Teste, was passiert, wenn ein Token aus zu vielen Zeichen besteht.
2. Mit dieser Assertion soll sichergestellt werden, dass es zu keinem undefinierten Verhalten kommt, wenn ein Token aus zu vielen Zeichen besteht:

   ```
   if (updateVal) {
       assert(lengthVal < sizeof(token.val) / sizeof(token.val[0]));
       token.val[lengthVal++] = ch;
       token.val[lengthVal] = 0;
   }
   ```

   Ist die Bedingung von assert nicht erfüllt, wird die Programmausführung mit einem Assertion-Failure beendet. Teste, ob das Problem damit wirklich gelöst wurde und korrigiere ansonsten den Code.

Schritt 8: Code Aufhübschen

Füge folgende Funktion in **lexer.abc** hinzu und verwende sie in der Implementierung von getToken. Eine geeignete Stelle für diese Funktion wäre am Anfang des Codes, nach den globalen Variablen und vor den Funktionen, die sie verwenden.

```
fn isDigit(ch: int): bool
{
    return ch >= '0' && ch <= '9';
}
```

Schritt 9: Identifier erkennen

Ein Identifier beginnt mit einem Buchstaben ('a' bis 'z' oder 'A' bis 'Z'), gefolgt von weiteren Buchstaben oder einer Ziffer ('0' bis '9'). Erweitere das Programm, sodass ein Identifier erkannt wird. Implementiere und verwende dafür eine Funktion isLetter und erweitere TokenKind um eine Enum-Konstante **IDENTIFIER**. Passe den Rest des Codes entsprechend an, um das neue Feature zu testen.

Schritt 10: Erster Kontakt mit "static"

Definiert man innerhalb einer Funktion eine Variable mit "static", dann hat sie die gleiche Lebensdauer wie eine globale Variable, ist aber nur innerhalb der Funktion sichtbar. Diese Variable hat also den gleichen Gültigkeitsbereich wie eine lokale Variable. Allerdings liegt diese Variable nicht auf dem Stack, sondern wird vom Compiler wie eine globale Variable behandelt. Sie befindet sich also im Datensegment oder im BSS-Segment. Wird eine static-Variable nicht explizit initialisiert, ist der Initialwert 0.

Mit folgender Anpassung kann durch die Verwendung einer static-Variable die Anzahl der globalen Variablen etwas reduziert werden:

```
// -- internal: only available within this translation unit

global ch: int;
global currentPos: Pos = {1, 0};

fn nextCh(updateVal: bool)
{
    static lengthVal: size_t;

    if (updateVal) {
        assert(lengthVal < sizeof(token.val) / sizeof(token.val[0]));
        token.val[lengthVal++] = ch;
        token.val[lengthVal] = 0;
    } else {
        lengthVal = 0;
    }
    ch = getchar();
    if (ch == '\n') {
        ++currentPos.row;
        currentPos.col = 0;
    } else {
        ++currentPos.col;
    }
}
```

Nun muss man die Implementierung von getToken entsprechend anpassen. Wenn Whitespaces ignoriert werden, übergibt man an **nextCh** das Argument false, ansonsten true. Das Zurücksetzen von **lengthVal** wird jetzt von **nextCh** übernommen:

```
fn getToken(): TokenKind
{
    while (ch == 0 || ch == ' ' || ch == '\n' || ch == '\t') {
        nextCh(false);
    }
    token.pos = currentPos;
    if (isDigit(ch)) {
        while (isDigit(ch)) {
            nextCh(true);
        }
        return token.kind = DECIMAL_LITERAL;
    // ..
}
```

Aufgabe: Passe das Programm entsprechend an, teste es und mache dich mit der neuen Implementierung vertraut.