C/C++ Materialpaket (Level AB)
07_IO – Input/Output

Prof. Dr. Carsten Link

Zusammenfassung

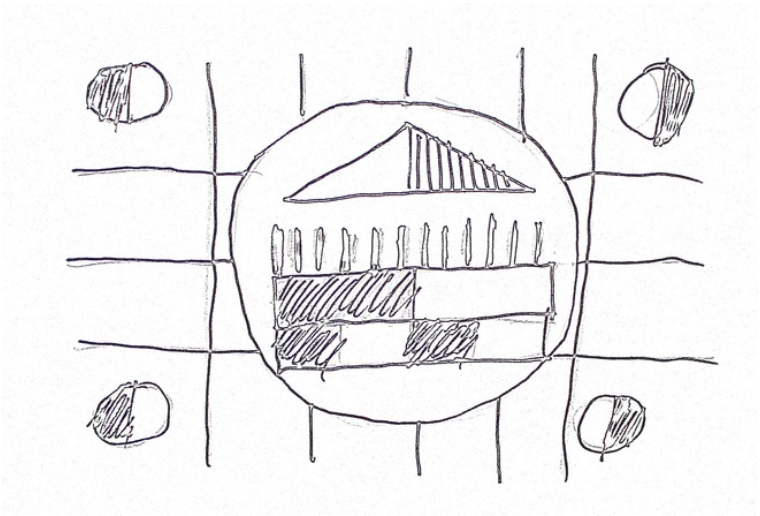

Abbildung 1: Testbild

Inhaltsverzeichnis

1 Kompetenzen und Lernegebnisse

Durch das Bearbeiten dieses Materialpaketes erwerben Sie diese Kompetenzen (Wissen, Fähigkeiten und Fertigkeiten zur Problemlösung):

Sie können mit Hilfe von speziellen Funktionen beeinflussen, wie Werte von benutzerdefinierten Datentypen erzeugt, geändert, kopiert und gelöscht werden.

Die oben genannten Kompetenzen erwerben Sie, indem Sie Lernziele erreichen, welche sich prüfen lassen. Lernegebnisse: Sie können nachweislich[1]:

- Möglichkeiten der Textausgabe benennen
- Fallstricke bei Text- und Binär-I/O benennen
- einfache Grammatiken definieren
- C++-Funktionen erstellen, um Texte erkennen, die einfachen Grammatiken genügen
- einen robusten Parser für `CSV`-Dateien erstellen

Hinweise:

- Kapitel, die mit (informativ) gekennzeichnet sind, sind nicht prüfungsrelevant und werden im Praktikum nicht benötigt.
- In diesem Materialpaket werden ignoriert: Character Encodings, Locales, i18n, L10n, effizientes on-demand Lesen
- Trennzeichen (Delimiters)

[1] Sie können das Erzielen der einzelnen Lernergebnisse beispielsweise bei einem Testat im Praktikum oder einer Aufgabe in der Modulprüfung nachweisen.

2 Konzepte

2.1 Output

2.1.1 Text ausgeben mit println() (informativ)

Die Template-Funktion `println()` ist in `println.hpp` definiert und verfügt über Spezialisierungen für eingebaute Typen. So wird beispielsweise bei Ausgaben ersichtlich, ob eine ausgegebene `5` von einem `int` oder einem `char` stammt (Werte 5 bzw. 53). Zur Formatierung werden Terminal-Steuersequenzen und Optionen der Standard-I/O-Streams verwendet (sie jeweils nachfolgende Abschnitte).

```
1 char c = 'c';
2 int i = -13;
3 double p = 355.0/113.0;
4 std::string s = "str";
5 println(c, " ", i, " ", p, " ", s);
```

Obiger Code gibt bei Ausführung Folgendes aus (im Terminal ist die Ausgabe formatiert: `-13` ist fett, `3.14159` ist unterstrichen):

```
$ ./a.out
c -13 3.14159 str
```

Soll ein Wert eines benutzerdefinierten Datentypens `T` ausgegeben werden, so muss dazu die Funktion `std::string as_string(T)` definiert sein. Hier ein Beispiel aus `src-cpp-student/04a_UDEF/RationalNumber.cpp`:

```
1 std::string as_string(RationalNumber r){
2   std::string result = "(";
3   result += std::to_string(r.zaehler);
4   result += "/";
5   result += std::to_string(r.nenner);
6   result += ")";
7   return result;
8 }
```

Fazit: `println()` ist sehr einfach zu benutzen, aber nur für einfache Problemstellungen geeignet.

Vorteile von `println()`

- einfacher Einstieg
- gut einsetzbar in der Lehre

Nachteile von `println()`

- nicht ausgereift
- Formatierung vorgegeben, nicht flexibel
- nur `stdout` unterstützt

2.1.2 Text ausgeben mit C++ I/O Streams (informativ)

Werte von `std::string` und allen eingebauten Datentypen, können mit den in der Standardbibliothek überladenen Shiftoperatoren `<<` ausgegeben werden. Die Standardbibliothek verbindet die Variablen `std::cout` und `std::cerr` mit den vom Betriebssystem bereitgestellten Ausgabekanälen 1 (`stdout`) und 2 (`stderr`).

```
char c = 'c';
int i = -13;
double p = 355.0/113.0;
std::string s = "str";
std::cout << c << " " << i << " " << p << " " << s << " " << std::endl;
std::cerr << "no error!" << std::endl;
```

Obiger Code gibt bei Ausführung Folgendes aus:

```
$ ./a.out_cout
c -13 3.14159 str
no error!
```

Soll ein Wert eines benutzerdefinierten Datentypens `T` ausgegeben werden, so muss dazu der Operator `std::ostream& operator<< (std::ostream& os, RationalNumber & T)` definiert sein. Hier ein Beispiel aus `src-cpp-student/04a_UDEF/RationalNumber.cpp`:

```
std::ostream& operator<< (std::ostream& os, RationalNumber &toBePrinted){
  os << "(" << toBePrinted.zaehler << "/"
  << toBePrinted.nenner << ")";
  return os;
}
```

Gleitkommawerte können formatiert werden, wie dieses Beispiel zeigt (für `T` werden Typen `double`, `float`, etc. angenommen):

```
// println.hpp
template <typename T>
std::string to_string_with_precision(const T & a_value)
{
  std::ostringstream out;
  if (   a_value  > 999999.9
      || a_value  < -999999.9
      || (a_value < 0.001 && a_value > 0.0)
      || (a_value >-0.001 && a_value < 0.0)
      ) {
    out << std::scientific;
  }else{
    out << std::defaultfloat;
  }
  out
```

```
  // << std::setprecision(n)
  // << std::setprecision(std::numeric_limits< double >::max_digits10 + 1)
  //<< std::setw(24)
  //<< std::defaultfloat
  //<< std::scientific
  //<< std::fixed
  << static_cast<long double>(a_value)  ;
  return out.str();
}
```

Die Klasse `std::stringstream` kann hilfreich sein, um Umwandlungen zwischen `std::string` und anderen Typen vornehmen zu lassen:

```
// https://cplusplus.com/reference/sstream/stringstream/stringstream/
// swapping ostringstream objects
#include <string>       // std::string
#include <iostream>     // std::cout
#include <sstream>      // std::stringstream

int main () {
  std::stringstream ss;
  ss << 100 << ' ' << 200;
  int foo,bar;
  ss >> foo >> bar;
  std::cout << "foo: " << foo << '\n';
  std::cout << "bar: " << bar << '\n';
  return 0;
}
```

Fazit: C++ `iostreams` sind immer verfügbar und flexibel, jedoch unhandlich.

Vorteile der C++ `iostreams`

- immer verfügbar
- vielfältige Formatierungsoptionen

Nachteile der C++ `iostreams`

- unhandlich, Zweckentfremdung von Operator Overloading

2.1.3 Text ausgeben mit fmtlib und std::fmt (informativ)

Die Bibliothek `{fmt}` (fmtlib, https://github.com/fmtlib/fmt) ist Grundlage der Standard Formatting library `std::format`. Beide weisen also viele Gemeinsamkeiten auf; `{fmt}` wird häufiger aufgewertet. Hier einige Beispiele:

```
// https://fmt.dev/latest/index.html
std::string s = fmt::format("The answer is {}.", 42);   // also: {:d}
fmt::print("Don't {}\n", "panic");
```

```
fmt::print(stderr, "System error code = {}\n", errno);
fmt::print("Hello, {name}! The answer is {number}. Goodbye, {name}.",
           fmt::arg("name", "World"), fmt::arg("number", 42));
```

Soll ein Wert eines benutzerdefinierten Datentypens `T` ausgegeben werden, so muss dazu ein `custom formatter`-Template definiert sein. Beispiele dazu finden sich unter https://en.cppreference.com/w/cpp/utility/format/formatter und https://www.modernescpp.com/index.php/extend-std-format-in-c-20-for-user-defined-types.

Vorteile von `{fmt}` und `std::format`

- modernes Interface (angelehnt an `format` von Python)
- schnell
- typsichere Ausgaben
- Unterstützung für Lokalisierung

Nachteile von `{fmt}` und `std::format`

- `std::fmt` erst ab C++20
- `{fmt}` ist externe Abhängigkeit

2.1.4 Formatierte Terminalausgabe (informativ)

Terminal-Emulatoren (wie `xterm` etc.) lassen sich mit speziellen Codes steuern. So lässt sich das Erscheinungsbild der Ausgaben beeinflussen. Die Steuercodes werden Escape-Sequenzen genannt. Hier einige Steuercodes, die von `println()` oder `AnsiConsole` verwendet werden:

```
// AnsiConsole.cpp
std::cout << "\033[0m\n";                      // reset attributes
std::cout << "\033[" << y << ";" << x <<  "f"; // move cursor to x,y
std::cout << (bold ? "\033[1;" : "\033[0;") << static_cast<int>(color) << "m";
std::cout << "\033[?25l";                      // hideCursor
std::cout << "\033[?25h";                      // showCursor
std::cout << "\033[2J";                        // clearScreen

// println.hpp
template <>
inline std::string as_string<double>(double d){
  return std::string("\033[4m") + to_string_with_precision(d) + std::string("\033[0m");
}
template <>
inline std::string as_string<char>(char c){
  return std::string("\033[3m") + std::string(1,c) + std::string("\033[0m"); // 1m,3m are g
}

template <>
```

```
20 inline std::string as_string<unsigned int>(unsigned int i){
21   return std::string("\033[1m") + to_string_with_precision(i) + std::string("\033[0m");
22 }
23 // see https://en.wikipedia.org/wiki/ANSI_escape_code
```

2.1.5 Binary Output (informativ)

Bei binärer Ausgabe ist oft das Dateiformat vorgegeben. In solchen Fällen eignet sich ein Ansatz, wie er in `src-cpp-student/03_DATA_b/typedMemory/typedMemory.cpp` zu sehen ist.

Kann das Dateiformat (oder das wire protocol im Netzwerk) selbst definiert werden, empfiehlt sich der Einsatz von Bibliotheken, wie beispielsweise Protocol Buffers.

Generell sollten diese Dinge beachtet werden:

- Eine Versionsnummer sollte untergebracht werden.
- Sicherheitsaspekte sollten mindestens mit Integritätssicherung und Verschlüsselung angegangen werden
- Typ- und Längeninformation ist den Daten voranzustellen.

2.2 Simple Input

2.2.1 Text einlesen mit C++ I/O Streams (informativ)

TBD

2.2.2 Text einlesen mit GNU readline() etc. (informativ)

Wird ohne Weiters von `std::cin` gelesen, so kann der Nutzer keine Cursortasten oder Backspace benutzen. Hier hilft eine Bibliothek wie GNU readline().

2.2.3 Textdateien einlesen mit C++ I/O Streams (informativ)

Das folgende Programm liest alle Zeilen einer Textdatei ein und speichert diese zu einem Gesamt-`std::string`, der am Ende ausgegeben wird. Damit wird die Benutzung der dazu nötigen Typen `std::ifstream` und `std::stringstream` sowie der Methode `std::getline` gezeigt.

```
1 // see https://stackoverflow.com/questions/58601912/c-reading-in-from-a-text-file
2
3 #include <fstream>
4 #include <string>
5 #include <iostream>
6 #include <sstream>
7
8 int main(int argc, char* argv[]){
```

```
9   std::string all_lines;
10  if (argc==2){
11    std::ifstream file(argv[1]);
12    //Read each line
13    std::string line;
14    while (std::getline(file, line))
15    {
16        std::stringstream ss(line);
17        all_lines += line + '\n';
18    }
19  } else {
20    std::cout << "usage: a.out file_name" << std::endl;
21  }
22
23  std::cout << "<" << all_lines << ">"  <<  std::endl;
24  return 0;
25 }
```

2.3 Input Parsing

Byte-Ströme, die von einem Programm eingelesen werden, wird Input genannt. Bei textuellem Input entsprechen die Bytes einem Character Encoding (beispielsweise `ISO-8859-15` oder `UTF-8`).

Das strukturierte Zerlegen von Input wird Parsing genannt (engl. *parse*: analysieren, zergliedern, verstehen). Demzufolge wird das Programm bzw. die Funktion, die diese Zerlegung vornimmt, Parser genannt.

Eine Analogie aus der Physik soll verdeutlichen, wie textueller Input strukturiert sein kann: Subatomare Teilchen (Elektronen, Neutronen, etc.) werden zu Atomen/Elementen kombiniert. Elemente lassen sich wieder Kombinieren zu Stoffen, zB. Eisen und Kohlenstoff zu Stahl, Kohlenstoff, Sauerstoff, Wasserstoff, Stickstoff zu Holz. Aus diesen Werkstoffen lassen sich Werkzeuge (Hammer) etc. bauen. Hierbei ist zu beachten, dass nicht alle mathematisch möglichen Kombinationen auch physikalisch existieren: Es existieren nur die Kombinationen von subatomaren Teilchen, die im Periodensystem der Elemte zu finden sind; ebenso lassen sich nicht beliebige Atome zu Molekülen kombinieren, da chemische Regeln eingehalten werden müssen. Weiterhin gibt es Stoffe die sich nicht zusammenbringen lassen.

Dieser Analogie folgend werden in diesem Materialpaket folgende Begriffe verwendet:

- `char`s entsprechen den subatomare Teilchen.
- **Syntaxelemente** stellen einfachste Sprachelemente (Literale, Schlüsselwörter, Interpunktion; z. B. in C++ `int`, `i`, `=`, `12` und `;`) dar. Syntaxelemente bestehen aus zulässigen Kombinationen von `char`s.

- **Syntaxmuster** bilden Syntaxkonstrukte bzw. Sprachkonstrukte (Funktionsdeklarationen, Kontrollfluss-Anweisungen; z. B. in C++ `void foo();`, `for( int k=0; k < 10; k++) m += k;`). Syntaxmuster bestehen aus zulässigen Kombinationen von Syntaxelementen (und ggf. anderen Syntaxmustern).

2.3.1 Regular Expressions

Now you have two problems.

TBD elaborate

2.3.2 EBNF-Grammatiken

Eine strukturierte Herangehensweise an die Probleme, die mit dem Einlesen von Texten einhergehen, ist:

1. eine Sprache zu identifizieren, der die einzulesenden Texte unterliegen
2. die Sprache in Form einer Grammatik festschreiben
3. aus der Grammatik Code herleiten, welcher Texte, die der Grammatik entsprechen, zuverlässig erkennen kann

Diese Herangehesnweise hat den Vorteil, dass sie recht einfach ist und daraus robuste Parser (Resultat von Punkt 3) entstehen. Im folgenden wird dieser Ansatz vereinfacht dargestellt. Es wird sich angelehnt an EBNF-Grammatiken welche in Form von recursive descent parsern umgesetzt werden.

Hier die Syntaxelemente von EBNF-artigen Grammatiken in einem Beispiel:

```
BITS ::= ['0b'] '0' ('0' | '1')*
```

Oben sind EBNF-Metazeichen zu sehen, welche folgende Bedeutungen haben:

- `::=` trennt Name der Regel von deren Definition
- `[]` Elemente sind optional (`0...1`)
- `|` oder
- `()*` Elemente kommen nicht oder mehrfach vor (`0...n`)
- `()+` Elemente kommen einmal oder mehrfach vor (`1...n`)

Demnach ergeben sich diese Bedingungen für gültige Bit-Zeichenketten:

- kann mit `0b` beginnen
- muss eine führende Null haben
- danach folgen beliebig viele Nullen oder Einsen

2.3.3 EBNF-Grammatiken für Syntaxelemente

Einfache Zusammensetzungen von `char`s werden im Folgenden Syntaxelemente genannt.

Zunächst die EBNF-artige Grammatik für einige Syntaxelemte:

```
// syntax elements

EXCL        ::= '!'
COMMA       ::= ','
LETTER      ::= 'a' | 'b' | 'c' ... | 'z' | 'A' | 'B' | 'C' ... | 'Z'
DIGIT       ::= '0' | '1' | '2' ... | '9'
DIGITS      ::= DIGIT (DIGIT)*
```

Beispielsweise legt die obige Regel `DIGITS` fest, wie Zahlliterale aufgebaut sind: beliebig viele Ziffer, aber mindestens eine.

In C++-Quellcode finden sich Schlüsselwörter (`int`, `struct`, etc.), Bezeichner (`main` etc.) und Literale verschiedener Datentypen (`23`, `3.1415`, `true` etc.), die sich mit ähnlichen Regeln beschreiben lassen, wie sie oben aufgeführt sind.

2.3.4 Überführung von Syntaxelement-Grammatiken in C++-Code

Es lässt sich leicht Code aus den Regeln ableiten, der Syntaxelemente akzeptiert bzw. erkennt (also einen Parser ableiten). Dazu können reguläre Ausdrücke verwendet werden. In diesem Materialpaket werden jedoch einfache C++-Funktionen verwendet, die alle die gleichen Parameter und Rückgabewerte haben:

```
1 int match_RULENAME(std::string src, int offset, int max_offset);
```

Es wird an eine solche Funktion ein String übergeben, in dem die Funktion der Regel entsprechend Zeichen erkennt. Die Funktion gibt zurück, wieviele Zeichen erkannt werden konnten.

Da alle `match_`-Funktionen uniform sind (also dieselben Parametertypen und -namen, und denselben Rückgabewert), ergeben sich diese Vorteile:

- Die `match_`-Funktionen lassen sich beliebig miteinander kombinieren.
- Änderungen in der Grammatik (zur Erweiterung oder Reparatur) lassen sich schnell umsezten, indem die Art der Kombination der `match_`-Funktionen verändert wird.
- Die Implementierungen sind besser lesbar.

Hier eine Funktion, die das Syntaxelement `DIGIT` umsetzt:

```
1  // DIGIT       ::= '0' | '1' | '2' | ... | '9'
2  int match_DIGIT(std::string src, int offset, int max_offset){
3    char c = src[offset];
4    if (        c == '0' ) {
5       return 1;
6    } else if ( c == '1' ) {
7       return 1;
8    } else if ( c == '2' ) {
9       return 1;
10   //            ...
```

```
  } else if ( c == '9' ) {
     return 1;
  } else {
     return 0;
  }
}
```

Alternativ lässt es sich auch kürzer implementieren: `return isdigit(src[offset]>0 ? 1 : 0);`

Die Funktion `match_DIGITS()` verwendet nun `match_DIGIT()`:

```
/// DIGITS      ::= DIGIT (DIGIT)*
int match_DIGITS(std::string src, int offset, int max_offset){
  if ( offset >= src.length() || max_offset >= src.length()) return 0;
  int match_len = 0;
  int is_digit = 0;
  do {
    is_digit = match_DIGIT(src, offset + match_len, max_offset);
    match_len += is_digit;
  } while (is_digit > 0);
  return match_len;
}
```

Die Funktion `match_DIGITS()` setzt also die Regel `DIGITS ::= DIGIT (DIGIT)*` um, indem sie zählt wieviele Zeichen ab der Stelle `offset` Ziffern sind. Ist bereits an der Stelle `src[offset]` keine Ziffer vorhanden, so wird `0` zurückgeliefert.

2.3.5 EBNF-Grammatiken für Syntaxmuster

Zusammensetzungen von Syntaxelementen werden im Folgenden Syntaxmuster genannt. Diese stellen komplexerer Sprachstrukturen dar.

Im folgenden werden drei Grammatikregeln und korrespondierender C++-Code gezeigt, die folgende Syntaxmuster erkennen:

1. `Hello,World!` (eine Feste Zeichenkette)
2. `Hello,James!` (ein Name)
3. `Hello,Bruce,Lemmy,Bon!` (eine Namensliste)

Die drei Regeln (`greet_world`, `greet_one`, `greet_many`) dieser etwas komplexer strukturierten Zeichenketten sehen wie folgt aus:

```
// Production Rules
// easy to map to specific C++-Functions
// may be recursive (usually inderectly; E)

greet_world ::= hello COMMA world EXCL
```

```
hello       ::= 'H' 'e' 'l' 'l' 'o'
world       ::= 'W' 'o' 'r' 'l' 'd'

greet_one   ::= hello COMMA name EXCL
name        ::= ident

greet_many  ::= hello COMMA name_list EXCL
name_list   ::= ident (comma ident)*
ident       ::= LETTER (LETTER | DIGIT)*
```

2.3.6 Überführung von Syntaxmuster-Grammatiken in C++-Code

Im folgenden werden einfache Handreichungen gegeben, mit denen sich Grammatiken in C++-Code überführen lassen, um einen Parser zu bauen.

Diese Regel beschreibt eine Folge:

```
sequence    ::= a b
```

Der C++-Code, der eine Folge erkennt, sieht so aus:

```
1  // sequence    ::= a b
2  int match_sequence(std::string src, int offset, int max_offset){
3    int match_len = 0;
4    int ml = 0;
5    len_a = match_a(src, offset, max_offset);
6    if ( len_a > 0 ) {
7      match_len += len_a;
8      int len_b = match_b(src, offset + match_len , max_offset);
9      if ( len_b > 0 ){
10       match_len += len_b;
11       return match_len;      // success!
12     } else {
13       return 0;              // no b
14     }
15   } else {
16     return 0;                // no a
17   }
18 }
```

Oder alternativ erkennen einer Folge mit early return:

```
1  // sequence    ::= a b
2  int match_sequence(std::string src, int offset, int max_offset){
3    int match_len = 0;
4    int ml = 0;
5
6    // recognise a
```

```
  len_a = match_a(src, offset, max_offset);
  if ( len_a == 0) {
    return 0;
  }
  match_len += len_a;

  // recognise b
  len_b = match_b(src, offset + match_len, max_offset);
  if ( len_b == 0) {
    return 0;
  }
  match_len += len_b;

  return match_len;
}
```

Diese Regel beschreibt eine Auswahl:

```
selection   ::= a | b
```

Der C++-Code, der eine Auswahl erkennt, sieht so aus:

```
// selection   ::= a | b
int match_sequence(std::string src, int offset, int max_offset){
  int len_a = match_a(src, offset , max_offset);
  int len_b = match_a(src, offset , max_offset);
  if( len_a > len_b ){
    return len_a;
  }else{
    return len_b;
  }
}
```

Diese Regel beschreibt eine optionale Wiederholung (`0...n`):

```
repetition   ::= ( a )*
```

Der C++-Code, der eine optionale Wiederholung erkennt, sieht so aus:

```
// repetition   ::= ( a )*
int match_repetition(std::string src, int offset, int max_offset){
  int match_len = 0;
  int len_a = 0;
  do {
    len_a = match_a(src, offset + match_len, max_offset);
    match_len += len_a;
  } while ( len_a > 0);
  return match_len;
```

```
}
```

Im folgenden wenden wir die Handreichungen auf die `hello`-Regeln an. Die Regel `greet_world` lässt sich sehr einfach in C++-Code überführen:

```
// greet_world ::= hello COMMA world EXCL
// hello       ::= 'H' 'e' 'l' 'l' 'o'
// world       ::= 'W' 'o' 'r' 'l' 'd'
int match_greet_world(std::string src, int offset, int max_offset){
  if ( offset >= src.length() || max_offset >= src.length()) return 0;
  int match_len = 0;
  int ml = 0;
  ml = match_hello(src, offset + match_len, max_offset);
  if(ml>0){                          // need 'Hello'
    match_len += ml;
    ml = match_COMMA(src, offset + match_len, max_offset);
    if(ml>0){                        // need ','
      match_len += ml;
      ml = match_world(src, offset + match_len, max_offset);
      if (ml>0){                     // need 'World'
        match_len += ml;
        ml = match_EXCL(src, offset + match_len, max_offset);
        if(ml>0){                    // need '!'
            match_len += ml;
            return match_len;        // finally ... success!
          } else {
            err("no '!' found");
            return 0;                // no '!' found
          }
      } else {
        err("no 'World' found");
        return 0;                    // no 'World' found
      }
    } else {
      err("no ',' found");
      return 0;                      // no ',' found
    }
  } else {
    err("no 'Hello' found");
    return 0;                        // no 'Hello' found
  }
}
```

Oben ist zu erkennen, dass die Regelsequenz durch geschachtelte `if`-Blöcke umgesetzt wurde, innerhalb derer die Funktionen `match_hello()`, `match_COMMA()`, `match_world()` und `match_EXCL()` nacheinander aufgerufen werden, um die

Länge des Gesamttreffers zu ermitteln.

Die Funktion `match_greet_one()` hat im Vergleich zu `match_greet_world()` nur eine kleine Änderung: es wird an einer Stelle `match_name()` statt `match_world()` aufgerufen (siehe `src-cpp-student/07_IO/z_match_patterns.cpp`).

Ebenso hat die Funktion `match_greet_many()` im Vergleich zu `match_greet_world()` nur eine kleine Änderung: es wird an einer Stelle `match_name_list()` statt `match_world()` aufgerufen. Die Funktion `match_name_list()` sieht so aus:

```
/// name_list   ::= name (comma name)*
int match_name_list(std::string src, size_t offset, size_t max_offset){
  if ( offset >= src.length() || max_offset >= src.length()) return 0;
  size_t match_len = 0;
  size_t ml = 0;
  do {
    ml = match_name(src, offset + match_len, max_offset);
    if(ml>0){
      std::string name = src.substr(offset + match_len, ml);
      log("mnl: found name '" + name + "'");
      match_len += ml;
      ml = match_COMMA(src, offset + match_len, max_offset);
      if(ml>0){
        match_len += ml;
      } else {
        log("mnl: end of list");
        return match_len;
      }
    } else {
      err("mnl: no name found");
      return 0;
    }

  } while (ml > 0);
  return match_len;
}
```

Die folgende Abbildung zeigt, welche Teile des Satzes `Hello,Bruce,Lemmy,Bon!` von welchen `match_`-Funktionen erkannt werden. Hier wird besonders deutlich, wie die Funktionen miteinander kombiniert wurden.

Das Problem des Zerlegen eines Textes wurde in viele kleine Teilprobleme zerlegt. Die Lösungen der Teilprobleme (die `match_`-Funktionen) lassen sich sehr einfach kombinieren, um die Regeln der Grammatik abzubilden.

3 Material zum aktiven Lernen

Regelmäßiger Hinweis: Da eine Programmiersprache nur durch aktive Verwendung erlernt werden kann, werden im Folgenden Aufgaben zum praktischen Üben vorgestellt. Zunächst wird ein Grundgerüst (C/C++Programm) erstellt, welches dann auf mehrere Arten modifiziert wird. Insbesondere die Modifikationen ermöglichen es dem Lernenden (und auch dem Lehrenden), die Qualität des Kompetenzerwerbs bzgl. dieses Materialpakets bewerten zu können.

3.1 Aufgabe: Grundgerüst

Nehmen Sie die Dateien `z_build.sh` und `z_match_main.cpp` sowie Code aus `z_main_ifstream.cpp` als Ausgangspunkt, um eigene Parser-Funktionen erstellen zu können.

1. Bauen Sie die Funktion `std::string preprocess(std::string)` ein, welche einen von Leerzeichen etc. befreite Zeichenkette liefert (siehe Tabelle unter `isspace()` auf `cppreference.com`). Stellen Sie einen Aufruf dieser Funktion dem Parser voran.
2. Erstellen Sie `match_`-Funktionen zu der in `z_test_s_expr.cpp` angegebenen Grammatik (eine pro Grammatikregel), mit denen sich Zeichenketten der Art `(+ 12 34)`, `(- 78 9)` und `(+ 1 2 3 456)` erkennen lassen. Stellen Sie sicher, dass Zeichenketten dieser Art nicht erkannt werden: `(+ 12)`, `(- 78 9` und `(1 2 3 456)`. Hinweis: es ist einfacher, wenn Sie mit der Implememtierung bei den Elementregeln (`DIGITS` etc.) anfangen und sich zu `S_EXPR ::=` hocharbeiten.

3.2 Aufgabe: Modifikationen

Regelmäßiger Hinweis: Weiter unten ist eine Liste mit Modifikationen gegeben, die zwei Zwecken dienen: 1) Sie dienen als Richtschnur für das Praktizieren und Üben der Inhalte dieses Materialpakets. 2) Die Modifikationen können im Rahmen eines Testats als Aufgabe verwendet werden, durch deren Lösung Studierende nachweisen können, dass sie den Stoff dieses Materialpakets beherrschen. Stellen Sie sicher, dass Sie jede einzelne der nachfolgenden Modifikationen innerhalb weniger Minuten (ca. 5 - 10) vor Zuschauern (Testatsituation) umsetzen können. Konkret sollen Sie im Testat in der Lage sein, das gegebene Grundgerüst um mindestens eine zufällig ausgewählte Modifikation zu erweitern. Bereiten Sie dazu auf ihrer Arbeitsumgebung ein Verzeichnis vor, welches ausschließlich das Grundgerüst enthält. **Arbeiten Sie also auf einer Kopie des Verzeichnisses `src-cpp-student/`!** Achten Sie darauf, dass der Text auf Ihrem Bildschirm in heller Umgebung aus einem Meter Abstand heraus gut lesbar ist (light mode, große Schrift).

Modifikationen:

1. Erstellen Sie eine Grammatik, die stark vereinfachte C-Funktionsdeklarationen beschreibt. Beispielsweise sollen diese Zeichenketten ihrer Grammatik genügen: `void foo(char c);` oder `int bar(double d, int i);`).
2. (Level C) Setzen Sie die Grammatik um in `match_c_decl()`.
3. Modifizieren Sie ihren Listenparser aus dem Grundgerüst so, dass statt Zahlen auch geklammerte Ausdrücke stehen können (z.B. `(+ 12 (- 78 9))`).
4. Erweitern Sie die Haupt-`match_`-Funktion aus der vorherigen Aufgabe um einen Parameter des Typs `std::vector<int>&` in dem alle gefundenen Zahlen gesammelt werden. Ggf. müssen Sie weitere `match_`-Funktion erwei-

tern, um die Vektorreferenz durchzureichen. Alternativ (Level C) lassen Sie ihren Parser das Ergebnis der Präfixausdrücke berechnen (Stichwort s-expressions von Lisp).

5. Erstellen Sie einen Parser, der CSV-Dateien (Comma Separated Values) erkennen kann. Werte können Zahlen oder Wörter sein (Leerzeichen und Anführungsstriche brauchen nicht berücksichtigt werden).
6. (Level C) Modifizieren Sie ihren CSV-Parser, dass Zeichenketten auch Leerzeichen enthalten können (aber kein Komma).
7. (Level C) Modifizieren Sie ihren CSV-Parser, so dass Werte optional in Anführungsstrichen stehen können.

3.3 Verständnisfragen

Nach Bearbeitung des Kapitels "Konzepte", der Erstellung des Grundgerüsts sowie dem Üben der Modifikationen sollten Sie in der Lage sein, die folgenden Fragen zu beantworten.

1. Was geschieht, wenn Sie `match_greet_one()` mit `"Hello,World!"` füttern? Finden Sie ein ähnliches Szenario bei C++-Compilern. Was fehlt unserem Parser in diesem Kontext?
2. (Level C) Was ist besonders an dieser Grammatik?

```
expr     ::= operand ( OPERATOR operand )*
OPERATOR ::= '+' | '-' | '*' | '/'
operand  ::= DIGITS | '(' expr ')'
```

3. Die `match_`-Funktionen haben alle dieselbe Signatur und geben die Trefferlänge zurück. Was ist der Vorteil dieser Uniformität?

4 Nützliche Links

- Railroad Diagram Generator, https://www.bottlecaps.de/rr/ui
- cplusplus.com: Input/output with files, https://cplusplus.com/doc/tutorial/files/
- Text Encoding:
  - International Components for Unicode, https://github.com/unicode-org/icu
  - iconvpp: Simple wrapper of iconv for C++, https://github.com/nonouno/iconvpp
- scnlib is a modern C++ library for replacing scanf and std::istream: https://github.com/eliaskosunen/scnlib und https://scnlib.readthedocs.io/en/v1.0/
- Protocol Buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data. https://protobuf.dev

5 Literatur

- Compilerbau: Eine Einführung von Wirth, Niklaus
- Compilers: Principles, Techniques, and Tools (red dragon book) by Aho, Sethi, Ullman
- [PPP] Stroustrup, Bjarne: Programming - Principles and Practice using C++
- [TCPL] Stroustrup, Bjarne: The C++ Programming Language, Fourth Edition