Informatik Übungsstunde - Woche 9

Eric Ceglie

Strings und Rekursion (fortgeschrittene Aufgabe)

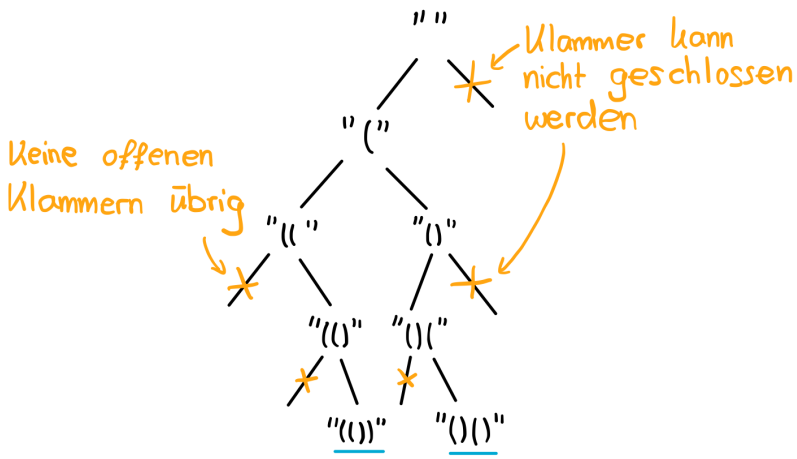
Im folgenden lösen wir eine etwas schwierigere, aber interessante Rekursionsaufgabe.

Aufgabe: Schreibe ein C++ Programm, das für eine gegebene Zahl $n \in \mathbb{N}$ alle möglichen Kombinationen von *mathematisch gültigen Klammerungen* mit n offenen und geschlossenen Klammern berechnet und ausgibt.

Beispiel:

```
Input:  n = 3
Output: ((())), (()()), (())(), ()(()), ()()()
```

Hinweis: Betrachte für den Input $n = 2$ den folgenden Rekursionsbaum:



Lösung:

```cpp
#include <iostream>
#include <vector>
#include <string>

void solver(std::vector<std::string>& res, int n, std::string current, int
opened, int closed){
    if(opened == n && closed == n){
        res.push_back(current);
        return;
    }
    if(opened < n){
```

```
        solve(res, n, current + "(", opened+1, closed);
    }
    if(closed < opened && closed < n){
        solve(res, n, current + ")", opened, closed+1);
    }
    return;
}

int main(){
  std::vector<std::string> result;
  int n;
  std::cin >> n;

  solver(result, n, "", 0, 0);

  for(std::string s : result){
    std::cout << s << "\n";
  }
  return 0;
}
```

EBNF

EBNF steht kurz für "Extended Backus-Naur form" und ist eine Methode Regeln zu definieren, die bestimmen, wie Elemente eines sogenannten Alphabets zusammen kombiniert werden dürfen, um einen gültigen Satz zu formen. Dabei treten oft rekursive Definitionen auf, die es erlauben eine komplexe formale Grammatik übersichtlich zu definieren und auch als C++ Programm zu implementieren.

Um eine EBNF zu definieren haben wir verschiedene Symbole zur verfügung:

Symbol	Verwendung
... | ...	Alternative (OR)
... ...	Aufzählung (AND)
{...}	Repitition (beliebig oft oder gar nie)
[...]	Repetition (maximal 1 Mal)
(...)	Gruppierung
... = ...	Definition (endet immer mit .)
"..."	Terminales Symbol

Beispiel

Wir wollen mit dem Alphabet $\{A, a, _\}$ und mit folgender Regel Sätze bauen: Das Symbol A darf nur am Anfang oder direkt nach dem Symbol _ erscheinen und das Symbol _ darf nicht mehrfach

hintereinander, am Anfang oder am Ende stehen.

Eine EBNF, die diese Regel formal definiert sieht wie folgt aus:

```
seq = term | term "_" seq
term = "A" | "A" lowerterm | lowerterm
lowerterm = "a" | "a" lowerterm
```

Zum Beispiel erhalten wir mit dieser Beschreibung den Satz *Aaa_A* wie folgt:

```
-> term "_" seq
-> "A" lowerterm "_" seq
-> "Aa" lowerterm "_" seq
-> "Aaa_" seq
-> "Aaa_" term
-> "Aaa_A"
```

Wichtige Funktionen zur Parser-Programmierung

```
// PRE:  gültiger Input Stream
// POST: gibt true zurück, falls weiterer Input vorhanden ist
//       und andernfalls false
bool input_available(std::istream& input);

// PRE:  gültiger Input Stream
// POST: der nächste character im stream wird zurückgegeben (ohne
//       aufgebraucht zu werden), falls kein weiterer Input vorhanden
//       ist wird 0 zurückgegeben
char peek(std::istream& input);

// POST: führende whitespaces (Abstände) werden vom Stream aufgebraucht
//       und der erste non-whitespace character wird ausgegeben (aber wird
//       nicht aufgebraucht), falls ein Fehler auftritt oder der Stream
//       endet wird 0 zurückgegeben
char lookahead(std::istream& input);

// PRE:  Gültiger Input Stream, ch > 0
// POST: Falls ch der nächste Charakter im Stream ist, wird dieser
//       aufgebraucht und true wird zurückgegeben. Falls nicht, wird
//       false zurückgegeben
bool consume(std::istream& input, char ch);
```

Structs

Ein Struct ist ein "selbstgebauter Typ" und ermöglicht somit das Erstellen von viel allgemeineren Variablen.

Beispiel

Mit einer Struct können wir zum Beispiel eine Datenbank für Informationen zu Studenten wie folt erstellen:

```
struct student{
        int alter;
        std::string name;
        unsigned int legi_nr;
        std::vector<double> noten;
};
```

Initialisierung:

```
student lisa = {20, "Lisa", 2194513, {4.5, 5.5, 5.25}};
```

Zugriff:

```
std::cout << lisa.legi_nr;
for(double note : lisa.noten){
        std::cout << note << "\n";
}
```

Damit können wir nun ganz einfach eine Datenbank erstellen:

```
std::vector<student> students = {
        {20, "Lisa", 2194513, {4.5, 5.5, 5.25}},
        {19, "Peter", 2065928, {3.5, 3.75}},
        {21, "Tim", 2190851, {5, 5.75, 6}}
        };
```