Praktikumstermin Nr. 07, INF: *Recursive Descent* Parser (Funktionen, Rekursion, Referenzparameter)

Die in dieser Anleitung vorgeschlagenen Arbeitsschritte sind wie gesagt nur ein Vorschlag, wie Sie sich schrittweise die Thematik und die zugehörigen Lösungen erarbeiten könnten. Was letztendlich zählt ist die funktionierende Gesamtlösung!

Ihre Vorbereitungen für diesen Praktikumstermin

Dieses Aufgabenblatt behandelt ein komplexeres Sachthema. Der resultierende C++ Code ist letztendlich eigentlich recht einfach aufgebaut, aber das Erschließen des Weges dorthin ist nicht einfach. Bereiten Sie sich daher gewissenhaft vor: **Sie müssen am Praktikumstermin auf jeden Fall ein klares Verständnis des zu lösenden Problems (Parsen von Texten) haben und möglichst auch die Lösung fertig programmiert haben**. Die Aufgabenstellung beschreibt sehr detailliert, was Sie zu programmieren haben. Sie müssen dies aber genau umsetzen und auch das zu lösende Problem verstanden haben.

Parser

Der Quelltext von C++ Programmen muss den Regeln der Programmiersprache C++ genügen. So muss z.B. jede Anweisung mit einem Semikolon abgeschlossen werden. Wird eine Anweisung im C++ Quelltext also nicht mit einem Semikolon abgeschlossen, so ist dies kein korrekter C++ Quelltext. Man nennt dies dann einen *Syntaxfehler* im C++ Quelltext (wurde auch in der GIP Vorlesung besprochen).

Ein *Parser* ist ein Computerprogramm, welches einen beliebigen Eingabetext einliest und dann bestimmt, ob dieser Eingabetext einem bestimmten Syntax-Regelwerk genügt. Ein bestimmter Parser ist immer genau für ein bestimmtes Regelwerk programmiert.

Sie werden jetzt einen Parser programmieren, welcher eine Zeichenkette einliest und prüft, ob diese Zeichenkette einem vorgegebenen Syntax-Regelwerk genügt.

Grammatiken

Die sogenannten *Grammatiken* sind maschinenlesbare Regelwerke, mit denen man Regeln für syntaktisch korrekte „strukturierte Texte"

beschreiben kann. So sind also z.B. die Regeln zur Bildung korrekter C++ Quelltexte über solch eine Grammatik definiert.

Innerhalb der C++ Grammatik sind einige Regeln zuständig für die Bildung syntaktisch korrekter Bedingungsausdrücke (für Verzweigungs- und Schleifenbedingungen), also für Vergleichsoperatoren, boolesche Operatoren sowie Klammerung. Diese Bedingungsausdrücke wollen wir im Weiteren betrachten.

Beispiele:

`4 > 3 && 10 == 11` ist ein korrekt gebildeter C++ Bedingungsausdruck.
`(4 > 3) && 10 == 11` ebenso.
`(4 > ) && (10 == 11` hingegen nicht.

Ein Parser für solche Bedingungsausdrücke würde also bestimmen, dass `4 > 3 && 10 == 11` sowie `(4 > 3) && (10 == 11)` syntaktisch korrekt gebildete Bedingungsausdrücke sind, hingegen `(4 > ) && (10 == 11` ein syntaktisch inkorrekter Bedingungsausdruck ist.

Der *resultierende Wert* eines Ausdrucks wird vom Parser nicht berechnet. D.h. ob die Bedingung zu `true` oder `false` auswertet, interessiert den Parser nicht und wird auch nicht von diesem berechnet. Der Parser kümmert sich nur um die Korrektheitsprüfung bezüglich des Syntax-Regelwerks.

Unsere Bedingungsausdruck-Grammatik

Unsere Bedingungsausdruck-Grammatik besteht aus mehreren Regeln und definiert gültige Bedingungsausdrücke über einstelligen Zahlen, den Vergleichsoperatoren **`>`** und **`<`**, den logischen Operatoren **`U`** (für „und“) und **`O`** (für „oder“) sowie Klammerung.
Die Schreibweise unserer Bedingungsausdrücke weicht also etwas von der C++ Schreibweise ab.
Als Besonderheit muss bei unseren Bedingungsausdrücken außerdem jeder korrekte Ausdruck mit einem Punkt abgeschlossen werden.

Beispiel-Bedingungsausdrücke für unser Regelwerk:

`4 > 3 U 10 < 11.` ist ein korrekt gebildeter Bedingungsausdruck.
`(7 < 4) O (10 > 11).` ebenso.
`(4 > ) U (10 < 11` hingegen nicht.

Aufgabe INF-07.01: (*Recursive-Descent* Parser; Parsen durch Rekursiven Abstieg)

Programmieren Sie einen Parser, welcher eine Zeichenkette `input` vom Datentyp `string` einliest und prüft, ob diese Zeichenkette den Regeln unserer Bedingungsausdruck-Grammatik genügt. Der Parser errechnet nicht den Ergebnis-Wert des Bedingungsausdrucks, sondern prüft nur dessen syntaktisch korrekten Aufbau. Das Ergebnis der Prüfung ergibt sich durch Analyse der Bildschirmausgaben Ihres Parsers (siehe Testlauf).

Ihr Parser soll u.a. zwei gegebene Hilfsfunktionen `expect()` und `match()` nutzen. Bei diesen Funktionen soll die Integer Variable `pos` die Position innerhalb der Eingabe-Zeichenkette speichern, an der sich der Parser bei seiner Analyse gerade befindet.

`expect()`: prüft nach, ob das nächste Zeichen aus `input` (d.h. das Zeichen an Position `pos`) einem bestimmten Zeichen entspricht. Mittels dieser Funktion entscheidet der Parser zwischen verschiedenen Regel-Alternativen der Grammatik. Die Position `pos` wird dabei nicht verändert, d-h- das Zeichen wird nur "angetestet", nicht wirklich "konsumiert". `expect()` prüft dabei auch, ob der Parser bei der Analyse von `input` schon das Ende der eingegebenen Zeichenkette erreicht hat. Ist dies der Fall, so liefert das „Antesten" irgendeines nächsten Zeichens natürlich immer ein negatives Ergebnis (sprich: `false`) zurück.

```
bool expect(char c, string &input, size_t pos)
{
    cout << "Teste auf das Zeichen " << c << endl;
    if ( pos >= input.length() )
    {
        cout << "Aber schon am Ende der "
             << "Eingabe-Zeichenkette angelangt.\n";
        return false;
    }
    if ( input.at(pos) == c )
    {
        cout << "Zeichen " << c << " gefunden.\n";
        return true;
    }
    else
    {
        cout << "Test auf " << c << " nicht erfolgreich. "
             << "Stattdessen " << input.at(pos)
             << " gesehen." << endl;
        return false;
    }
```

```
}
```

Arbeitsschritt 2: *(siehe auch zugehörige Dateien in Ilias)*

Nutzen Sie das C++ Programm aus Ilias, um die Funktionsweise der vorgegebenen match() Funktion durch die Beschäftigung mit dem ausführbaren C++ Code zu verstehen.

In diesem Arbeitsschritt ist noch keine C++ Programmierung ihrerseits erforderlich. Studieren Sie (und modifizieren Sie ggfs.) nur die vorgegebenen Testfälle.

`match()`: „konsumiert" das nächste Zeichen aus `input` (d.h. das Zeichen an Position `pos`) in der Erwartung, dass es sich um das Zeichen `c` handeln muss. "Konsumieren" bedeutet dabei, dass die Position `pos` um eine Position erhöht (weiter nach rechts gesetzt) wird und damit an die Position hinter dem erwarteten Zeichen gesetzt wird.
Die Erwartungshaltung ergibt sich aus den Regeln der Grammatik. Entspricht das nächste Zeichen nicht der Erwartung, so ist die `input` Zeichenkette nicht entsprechend den Regeln der Grammatik aufgebaut.

```
void match(char c, string &input, size_t &pos)
{
    cout << "Betrete match() fuer das Zeichen " << c << endl;
    if ( pos >= input.length() )
    {
        cout << "Error! Input-Zeichenkette zu kurz. "
             << "Erwarte noch das Zeichen " << c << endl;
        cout << "Verlasse match()" << endl;
        return;
    }
    if ( input.at(pos) != c )
    {
        cout << "Error! An Position "
             << pos << " erwarte "
             << c << " und sehe " << input.at(pos) << endl;
        cout << "Verlasse match()" << endl;
        return;
    }
    pos++;
    cout << "Zeichen " << c << " konsumiert.\n"
         << "Verlasse match() fuer das Zeichen "
         << c << endl;
}
```