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

Arbeitsschritt 8: *(siehe auch zugehörige Dateien in Ilias)*

Programmieren Sie nun die parse_gesamtausdruck() Funktion. Nutzen Sie in ihrem Code die parse_xxx() Funktionen, die Sie in vorigen Arbeitsschritten programmiert haben. Nutzen Sie das C++ Programm aus Ilias, um ihre neue parse_gesamtausdruck() Funktion mittels des ausführbaren C++ Codes zu testen. Wandeln Sie dann das Hauptprogramm main() in das ursprünglich vorgegebene main() des Aufgabenblatts um.

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.

Im Folgenden werden die einzelnen Syntax-Regeln unseres Bedingungs-ausdruck-Syntax-Regelwerks beschrieben.

Regel: `Number`
Eine `Number` besteht entweder aus der Ziffer `'0'` oder der Ziffer `'1'` oder … oder der Ziffer `'9'`.

```
Number  ::=  '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
```

Regel: `Operand`
Ein `Operand` besteht entweder aus einer `Number`, oder er besteht aus dem Zeichen `(` gefolgt von einem `Ausdruck` (siehe Regel für `Ausdruck` weiter unten) gefolgt von dem Zeichen `)`, d.h. der runden schließenden Klammer.

```
Operand  ::=  Number
           |  '('  Ausdruck  ')'
```

Regel: `Term`
Ein `Term` besteht aus `Operand`*en*, die mit dem Zeichen `'>'` oder dem Zeichen `'<'` verknüpft sind.
Die **`(`** … **`)`** Klammern im Regelwerk sind nötig, um zu klären, worauf sich die „oder" Operation `|` des Regelwerks bezieht.
Die geschweiften Klammern **`{`** … **`}`** bedeuten „keine, eine oder mehrere Wiederholungen" dessen, was in den Klammern enthalten ist, also z.B. `4 > 3 > 2 < 8`.

```
Term  ::=  Operand { ( '>' | '<' ) Operand }
```

Regel: `Ausdruck`
Ein `Ausdruck` besteht aus `Term`*en*, die mit dem Zeichen `'U'` oder dem Zeichen `'O'` verknüpft sind. Z.B. `4 > 3 > 2 U 5 > 4 O 2 < 3`

```
Ausdruck  ::=  Term { ( 'U' | 'O' ) Term }
```

Regel: `Gesamtausdruck`
Ein `Gesamtausdruck` besteht aus einem `Ausdruck`, gefolgt von dem Zeichen `'.'`.

```
Gesamtausdruck  ::=  Ausdruck '.'
```

Damit sieht unsere Grammatik für gültige Gesamtausdrücke wie folgt aus:

```
Gesamtausdruck ::= Ausdruck '.'

Ausdruck ::= Term { ( 'U' | 'O' ) Term }

Term     ::= Operand { ( '>' | '<' ) Operand }

Operand  ::= Number
           | '(' Ausdruck ')'

Number   ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
```

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).

Beispiel:

Bei Eingabe von `4 > 3 U 10 < 11.` soll der Parser bestimmen, dass es sich um einen syntaktisch korrekten Bedingungsausdruck handelt. Bei Eingabe von `(4 > ) U (10 < 11` soll der Parser bestimmen, dass der eingegebene Bedingungsausdruck syntaktisch inkorrekt ist.

Details der Eingabe und Ausgabe Ihres Parsers siehe später bei den Testläufen.

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;
    }
}
```

`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;
}
```

`main()`: das Hauptprogramm Ihres Parsers.

```
int main()
{
    size_t pos = 0;
    string input = "";
    cout << "Bitte geben Sie die Zeichenkette ein: ";
    getline(cin, input);
    parse_gesamtausdruck(input, pos);
    if ( pos != input.length() )
        cout << "Error! Noch Input-Zeichen uebrig.\n";
    system("PAUSE");
    return 0;
}
```

Von Ihnen zu programmierende Funktionen des Parsers

Schreiben Sie *zu jeder Regel* der Grammatik *eine C++ Funktion* für den Parser. Gehen Sie dabei nach folgendem Schema vor:

Für jede Regel *Name* `::=` … schreiben Sie eine C++ Funktion `void parse_`*name*`(string &input, size_t &pos);` wobei Sie *name* durch den jeweiligen Regelnamen ersetzen.

Also z.B. eine C++ Funktion `parse_number()` für die Regel `Number ::=` …

Schreiben Sie an den Beginn jeder dieser Funktionen eine Ausgabe
`cout << "Betrete parse_`*name*`()" << endl;`
und ans Ende sowie vor jede `return` Anweisung innerhalb der Funktion eine Zeile
`cout << "Verlasse parse_`*name*`()" << endl;`

Jedes Vorkommen eines Regelnamens auf einer rechten Seite einer Regel wird in einen C++ Funktionsaufruf der entsprechenden Funktion umgewandelt. So wird z.B. aus
`Gesamtausdruck ::= Ausdruck …`
ein

```
void parse_gesamtausdruck (string &input, size_t &pos)
{
      …
      parse_ausdruck(input, pos);
      …
```

Jedes Vorkommen eines konkreten Zeichens auf der rechten Seite einer Regel wird in einen C++ Funktionsaufruf `match()` mit dem entsprechenden Zeichen als erstes Argument umgewandelt. So wird z.B. aus `'+'` ein `match('+', input, pos)`

Sollte eine Regel der Grammatik mehrere alternative rechte Seiten haben (siehe Regeln `Number` und `Operand`), so ist das zugehörige C++ Konstrukt im Rumpf der Funktion `parse_name()` eine …

```
if { … } else if { … } else { … }
```

… Verzweigung.

Testen Sie zuerst auf die Fälle, die mit einem konkreten Input-Zeichen beginnen (dies ist für `Operand` relevant). So wird z.B. aus

```
Operand ::= … | '(' Ausdruck …
```

ein …

```
void parse_operand(string &input, size_t &pos)
{
      …
      if ( expect('(', input, pos) )
     {
          cout << "Betrete parse_operand(): '(' Fall" << endl;
          match('(', input, pos);
          parse_ausdruck(input, pos);
           …
```

Behandeln Sie den Fall ohne konkretes Input-Zeichen am Anfang im `else` Zweig.

Kommt in einer Regel der Grammatik auf der rechten Seite ein „keine, eine oder mehrere Wiederholungen" Konstrukt **{** … **}** vor, so ist das zugehörige C++ Konstrukt im Rumpf der Funktion `parse_name()` eine `while` Schleife. So wird z.B. aus

```
Ausdruck ::= … { ( 'U' | 'O' ) Term }
```

ein …

```
void parse_ausdruck(string &input, size_t &pos)
{
      …
     while ( expect('U', input, pos) ||
             expect('O', input, pos) )
     {
          if ( expect('U', input, pos) )
          {
               cout << "Betrete parse_ausdruck(): 'U' Fall"
                    << endl;
               …
               cout << "Verlasse parse_ausdruck(): 'U' Fall"
                    << endl;
          }
          else if ( expect('O', input, pos) )
          {
               cout << "Betrete parse_ausdruck(): 'O' Fall"
                    << endl;
               …
```

```
            cout << "Verlasse parse_ausdruck(): 'O' Fall"
                 << endl;
        }
    }
    …
}
```

Ein Parser lässt sich übrigens nur dann nach dem obigen Vorgehen konstruieren, wenn die zugehörige Grammatik eine bestimmte Form hat. Dieses Vorgehen funktioniert also nicht für jede beliebige Grammatik. Aber diese Aspekte werden wir hier (und in GIP generell) nicht weiter diskutieren.

Testlauf (Benutzereingaben unterstrichen):

```
Bitte geben Sie die Zeichenkette ein: (4>3)U2<7.
Betrete parse_gesamtausdruck()
Betrete parse_ausdruck()
Betrete parse_term()
Betrete parse_operand()
Teste auf das Zeichen (
Zeichen ( gefunden.
Betrete parse_operand(): '(' Fall
Betrete match() fuer das Zeichen (
Zeichen ( konsumiert.
Verlasse match() fuer das Zeichen (
Betrete parse_ausdruck()
Betrete parse_term()
Betrete parse_operand()
Teste auf das Zeichen (
Test auf ( nicht erfolgreich. Stattdessen 4 gesehen.
Betrete parse_number()
Teste auf das Zeichen 0
Test auf 0 nicht erfolgreich. Stattdessen 4 gesehen.
Teste auf das Zeichen 1
Test auf 1 nicht erfolgreich. Stattdessen 4 gesehen.
Teste auf das Zeichen 2
Test auf 2 nicht erfolgreich. Stattdessen 4 gesehen.
Teste auf das Zeichen 3
Test auf 3 nicht erfolgreich. Stattdessen 4 gesehen.
Teste auf das Zeichen 4
Zeichen 4 gefunden.
Betrete match() fuer das Zeichen 4
Zeichen 4 konsumiert.
Verlasse match() fuer das Zeichen 4
Verlasse parse_number()
Verlasse parse_operand()
Teste auf das Zeichen >
Zeichen > gefunden.
Teste auf das Zeichen >
Zeichen > gefunden.
```

```
Betrete parse_term(): '>' Fall
Betrete match() fuer das Zeichen >
Zeichen > konsumiert.
Verlasse match() fuer das Zeichen >
Betrete parse_operand()
Teste auf das Zeichen (
Test auf ( nicht erfolgreich. Stattdessen 3 gesehen.
Betrete parse_number()
Teste auf das Zeichen 0
Test auf 0 nicht erfolgreich. Stattdessen 3 gesehen.
Teste auf das Zeichen 1
Test auf 1 nicht erfolgreich. Stattdessen 3 gesehen.
Teste auf das Zeichen 2
Test auf 2 nicht erfolgreich. Stattdessen 3 gesehen.
Teste auf das Zeichen 3
Zeichen 3 gefunden.
Betrete match() fuer das Zeichen 3
Zeichen 3 konsumiert.
Verlasse match() fuer das Zeichen 3
Verlasse parse_number()
Verlasse parse_operand()
Verlasse parse_term(): '>' Fall
Teste auf das Zeichen >
Test auf > nicht erfolgreich. Stattdessen ) gesehen.
Teste auf das Zeichen <
Test auf < nicht erfolgreich. Stattdessen ) gesehen.
Verlasse parse_term()
Teste auf das Zeichen U
Test auf U nicht erfolgreich. Stattdessen ) gesehen.
Teste auf das Zeichen O
Test auf O nicht erfolgreich. Stattdessen ) gesehen.
Verlasse parse_ausdruck()
Betrete match() fuer das Zeichen )
Zeichen ) konsumiert.
Verlasse match() fuer das Zeichen )
Verlasse parse_operand(): '(' Fall
Verlasse parse_operand()
Teste auf das Zeichen >
Test auf > nicht erfolgreich. Stattdessen U gesehen.
Teste auf das Zeichen <
Test auf < nicht erfolgreich. Stattdessen U gesehen.
Verlasse parse_term()
Teste auf das Zeichen U
Zeichen U gefunden.
Teste auf das Zeichen U
Zeichen U gefunden.
Betrete parse_ausdruck(): 'U' Fall
Betrete match() fuer das Zeichen U
Zeichen U konsumiert.
Verlasse match() fuer das Zeichen U
Betrete parse_term()
Betrete parse_operand()
```

```
Teste auf das Zeichen (
Test auf ( nicht erfolgreich. Stattdessen 2 gesehen.
Betrete parse_number()
Teste auf das Zeichen 0
Test auf 0 nicht erfolgreich. Stattdessen 2 gesehen.
Teste auf das Zeichen 1
Test auf 1 nicht erfolgreich. Stattdessen 2 gesehen.
Teste auf das Zeichen 2
Zeichen 2 gefunden.
Betrete match() fuer das Zeichen 2
Zeichen 2 konsumiert.
Verlasse match() fuer das Zeichen 2
Verlasse parse_number()
Verlasse parse_operand()
Teste auf das Zeichen >
Test auf > nicht erfolgreich. Stattdessen < gesehen.
Teste auf das Zeichen <
Zeichen < gefunden.
Teste auf das Zeichen >
Test auf > nicht erfolgreich. Stattdessen < gesehen.
Teste auf das Zeichen <
Zeichen < gefunden.
Betrete parse_term(): '<' Fall
Betrete match() fuer das Zeichen <
Zeichen < konsumiert.
Verlasse match() fuer das Zeichen <
Betrete parse_operand()
Teste auf das Zeichen (
Test auf ( nicht erfolgreich. Stattdessen 7 gesehen.
Betrete parse_number()
Teste auf das Zeichen 0
Test auf 0 nicht erfolgreich. Stattdessen 7 gesehen.
Teste auf das Zeichen 1
Test auf 1 nicht erfolgreich. Stattdessen 7 gesehen.
Teste auf das Zeichen 2
Test auf 2 nicht erfolgreich. Stattdessen 7 gesehen.
Teste auf das Zeichen 3
Test auf 3 nicht erfolgreich. Stattdessen 7 gesehen.
Teste auf das Zeichen 4
Test auf 4 nicht erfolgreich. Stattdessen 7 gesehen.
Teste auf das Zeichen 5
Test auf 5 nicht erfolgreich. Stattdessen 7 gesehen.
Teste auf das Zeichen 6
Test auf 6 nicht erfolgreich. Stattdessen 7 gesehen.
Teste auf das Zeichen 7
Zeichen 7 gefunden.
Betrete match() fuer das Zeichen 7
Zeichen 7 konsumiert.
Verlasse match() fuer das Zeichen 7
Verlasse parse_number()
Verlasse parse_operand()
Verlasse parse_term(): '<' Fall
```

```
Teste auf das Zeichen >
Test auf > nicht erfolgreich. Stattdessen . gesehen.
Teste auf das Zeichen <
Test auf < nicht erfolgreich. Stattdessen . gesehen.
Verlasse parse_term()
Verlasse parse_ausdruck(): 'U' Fall
Teste auf das Zeichen U
Test auf U nicht erfolgreich. Stattdessen . gesehen.
Teste auf das Zeichen O
Test auf O nicht erfolgreich. Stattdessen . gesehen.
Verlasse parse_ausdruck()
Betrete match() fuer das Zeichen .
Zeichen . konsumiert.
Verlasse match() fuer das Zeichen .
Verlasse parse_gesamtausdruck()
Drücken Sie eine beliebige Taste . . .
```