Exercise Week 10

GianAndrea Müller
mailto:muellegi@student.ethz

May 9, 2018

Exercise Week 10

GianAndrea Müller
mailto:muellegi@student.ethz

May 9, 2018

Time Schedule

- 20' Backus-Naur-Form mit Übung
- 10' Datenstrukturen
- 5' Funktionsüberladung
- 10' Operatorüberladung
- 5' Konstante Referenzen

Time Schedule

- 20' Backus-Naur-Form mit Übung
- 10' Datenstrukturen
- 5' Funktionsüberladung
- 10' Operatorüberladung
- 5' Konstante Referenzen

Learning Objectives
Verständnis der EBNF
Kenntnis von Datenstrukturen und Funktionsüberladungen

Learning Objectives

- Verständnis der EBNF
- Kenntnis von Datenstrukturen und Funktionsüberladungen

Backus-Naur-Form

BNF
Die BNF ist eine formale Metasprache, die benutzt wird, um kontextfreie Grammatiken darzustellen.

Metasprache
Eine Metasprache ist eine "Sprache über Sprache".

Kontextfreie Grammatik
Eine kontextfreie Grammatik besteht aus Regeln die unabhängig vom Kontext angewandt werden können.

EBNF Metasprache Kontextfreie Grammatik

Backus-Naur-Form

BNF

Die BNF ist eine formale Metasprache, die benutzt wird, um kontextfreie Grammatiken darzustellen.

Metasprache

Eine Metasprache ist eine "Sprache über Sprache".

Kontextfreie Grammatik

Eine kontextfreie Grammatik besteht aus Regeln die unabhängig vom Kontext angewandt werden können.

EBNF Metasprache Kontextfreie Grammatik

Backus-Naur-Form
Kurz und simpel
Die Backus-Naur-Form ist eine Sprache die mit einfacher Syntax beschreibt, welche Sätze mit den Wörtern einer Sprache gebildet werden dürfen.
Aufbau
Alphabet = Terminalsymbole
Satzbau = Produktionsregeln = Nichtterminalsymbol

```
1 ZifferAusserNull = "1" | "2" | "3" | "4" |
      "5" | "6" | "7" | "8" | "9" ;
2 Ziffer = "0" | ZifferAusserNull ;
```

Backus-Naur-Form

Kurz und simpel

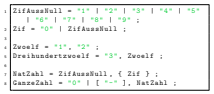
Die Backus-Naur-Form ist eine Sprache die mit einfacher Syntax beschreibt, welche Sätze mit den Wörtern einer Sprache gebildet werden dürfen.

Aufbau

- Alphabet = Terminalsymbole
- Satzbau = Produktionsregeln = Nichtterminalsymbol

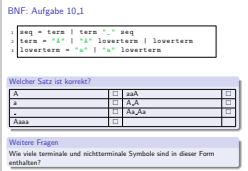
```
1 ZifferAusserNull = "1" | "2" | "3" | "4" |
      "5" | "6" | "7" | "8" | "9" ;
2 Ziffer = "0" | ZifferAusserNull ;
```

Erweiterte Backus-Naur-Form: Beispiel



Erweiterte Backus-Naur-Form: Beispiel

```
1 ZifAussNull = "1" | "2" | "3" | "4" | "5"
    | "6" | "7" | "8" | "9" ;
2 Zif = "0" | ZifAussNull ;
3
4 Zwoelf = "1", "2" ;
5 Dreihundertzwoelf = "3", Zwoelf ;
6
7 NatZahl = ZifAussNull, { Zif } ;
8 GanzeZahl = "0" | [ "-" ], NatZahl ;
```


BNF: Aufgabe 10_1

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

Welcher Satz ist korrekt?

A	☐	aaA	☐
a	☐	A_A	☐
_	☐	Aa_Aa	☐
Aaaa	☐		

Weitere Fragen

Wie viele terminale und nichtterminale Symbole sind in dieser Form enthalten?

BNF: Aufgabe 10_1

```
1 seq = term | term "_" seq
2 term = "A" | "A" lowerterm | lowerterm
3 lowerterm = "a" | "a" lowerterm
```

Welcher Satz ist korrekt?

A	☐	aaA	☐
a	☐	A_A	☐
_	☐	Aa_Aa	☐
Aaaa	☐		

Weitere Fragen

Wie viele terminale und nichtterminale Symbole sind in dieser Form enthalten?

BNF: Lösung 10_1

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

Welcher Satz ist korrekt?

A	☑	aaA	☒
a	☑	A_A	☑
_	☒	Aa_Aa	☑
Aaaa	☑		

Weitere Fragen

Es sind 3 terminale Symbole ("a", "A", "_") und drei nichtterminale Symbole ("seq", "term", "lowerterm") enthalten

BNF: Lösung 10_1

```
1 seq = term | term "_" seq
2 term = "A" | "A" lowerterm | lowerterm
3 lowerterm = "a" | "a" lowerterm
```

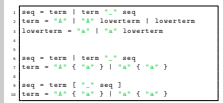
Welcher Satz ist korrekt?

A	☑	aaA	☒
a	☑	A_A	☑
_	☒	Aa_Aa	☑
Aaaa	☑		

Weitere Fragen

Es sind 3 terminale Symbole ("a", "A", "_") und drei nichtterminale Symbole ("seq", "term", "lowerterm") enthalten

Vorteile der EBNF

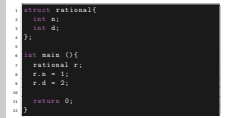


Vorteile der EBNF

- Die erweiterte BNF fuehrt zusaetzlich geschweifte Klammern ein. Diese bedeuten eine beliebige Anzahl Wiederholungen des enthaltenen Elements.
- Weiterhin werden eckige Klammer eingefuehrt. Diese bedeuten, dass der Inhalt 0 oder 1 mal eingefügt werden kann. Somit ist das eine Alternative.

```
1  seq = term | term "_" seq
2  term = "A" | "A" lowerterm | lowerterm
3  lowerterm = "a" | "a" lowerterm
4
5
6  seq = term | term "_" seq
7  term = "A" { "a" } | "a" { "a" }
8
9  seq = term [ "_" seq ]
10 term = "A" { "a" } | "a" { "a" }
```

struct



struct

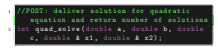
```c
struct rational{
  int n;
  int d;
};

int main (){
  rational r;
  r.n = 1;
  r.d = 2;

  return 0;
}
```

struct - Direkte Instantiierung

struct - Direkte Instantiierung

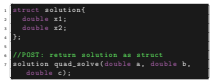
```c
struct rational{
  int n;
  int d;
}r,s;

int main (){
  r.n = 1;
  r.d = 2;

  return 0;
}
```

struct - Als Funktionsargument



struct - Als Funktionsargument

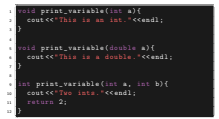
```
1 //POST: deliver solution for quadratic
    equation and return number of solutions
2 int quad_solve(double a, double b, double
    c, double & x1, double & x2);
```



struct - Als Funktionsargument

```
1 struct solution{
2   double x1;
3   double x2;
4 };
5
6 //POST: return solution as struct
7 solution quad_solve(double a, double b,
     double c);
```

Funktionsüberladung



Für Enthusiasten

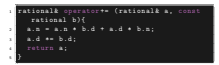
- Funktionsueberladung funktioniert mit Unterscheidung durch Anzahl Argumente und durch Typ der Argumente.
- Funktionsueberladung funktioniert weder mit Unterscheidung durch Rueckgabetyp noch durch Variablennamen.
- Ueberladungen dürfen andere Rueckgabetypen haben!

Funktionsüberladung

```
1  void print_variable(int a){
2    cout<<"This is an int."<<endl;
3  }
4
5  void print_variable(double a){
6    cout<<"This is a double."<<endl;
7  }
8
9  int print_variable(int a, int b){
10   cout<<"Two ints."<<endl;
11   return 2;
12 }
```

Für Enthusiasten

Operatorüberladung



Schönes Tutorial, Beispiele am Ende der Seite

- Jetzt da wir eigene Datentypen machen koennen lohnt es sich Operatoren neu zu definieren, damit sie auch fuer unsere speziellen Datentypen funktionieren.
- Funktionstyp per Konvention: Zuweisungsoperatoren geben eine Referenz zurueck! Somit ist eine Verkettung von Zuweisungen moeglich.
- Funktionsname: Schluesselwort operator und danach der Operator.
- Funktionsargumente: Binaerer Operator, hat 2 Argumente: Dasjenige auf der linken Seite und dasjenige auf der rechten Seite. Sie werden in dieser Reihenfolge ueberreicht.

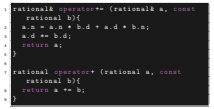
Operatorüberladung

```
1 rational& operator+= (rational& a, const
     rational b){
2   a.n = a.n * b.d + a.d * b.n;
3   a.d *= b.d;
4   return a;
5 }
```

$$\frac{a_n}{a_d} \leftarrow \frac{a_n}{a_d} + \frac{b_n}{b_d} = \frac{a_n \cdot b_d}{a_d \cdot b_d} + \frac{b_n \cdot a_d}{b_d \cdot a_d}$$

Schönes Tutorial, Beispiele am Ende der Seite

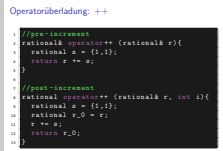
Operatorüberladung



- Rueckgabetyp hier ist ein Wert, keine Referenz.
- Grund dafuer ist, dass der Ueberladene += Operator wiederverwendet werden soll. Damit das keinen Einfluss auf a hat, wird a als Wert uebergeben und dann mit dem += lokal veraendert.
- So hat man nur eine Implementation fuer 2 Operatoren und eine Veraenderung des zugrundeliegenden Typen ist nicht so aufwaendig.

Operatorüberladung

```
1  rational& operator+= (rational& a, const
       rational b){
2    a.n = a.n * b.d + a.d * b.n;
3    a.d *= b.d;
4    return a;
5  }
6
7  rational operator+ (rational a, const
       rational b){
8    return a += b;
9  }
```



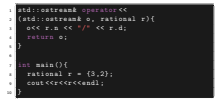
Operatorüberladung: ++

```
1  //pre-increment
2  rational& operator++ (rational& r){
3    rational s = {1,1};
4    return r += s;
5  }
6
7  //post-increment
8  rational operator++ (rational& r, int i){
9    rational s = {1,1};
10   rational r_0 = r;
11   r += s;
12   return r_0;
13 }
```

- In seiner grundlegenden Implementation erlaubt das pre-increment Verkettung. Das post-increment erlaubt das nicht. Beruhend auf dieser Tatsache werden diese Ueberladungen ausgelegt. Das fuehrt zu folgendem:
- Das pre-increment gibt eine Referenz zurueck (Verkettung), und nimmt als Parameter eine Referenz auf, da das Ziel der Operation eine Veraenderung des aufrufenden Parameters ist.
- Das post-increment gibt einen Wert zurueck (keine Verkettung), und nimmt als Parameter ebenfalls eine Referenz auf, da wiederum das Ziel der Operation eine Veraenderung des aufrufenden Parameters ist.
- Es kann im speziellen keine Referenz zurueckgegeben werden, da der Sinn des post-increments darin liegt den unveraenderten Wert der Variable zurueckzugeben, welcher nur als lokale Variable zur Verfuegung steht.
- Zur Unterscheidung der beiden Funktionen (im Prinzip Ueberladungen) erhaelt das post-increment einen Dummyparameter int i.
- some more content content

- Beim Aufruf erhaelt der ¡¡-Operator einen outstream von links und einen r-value von rechts.
- Es muss wieder eine Referenz auf den aufrufenden outstream zurueckgegeben werden, damit eine Verkettung mehrerer ¡¡ moeglich ist.

Operatorüberladung: <<



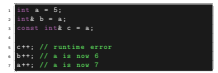
Operatorüberladung auf codeboard Operatorpräzedenz

Operatorüberladung: <<

```
1  std::ostream& operator<<
2  (std::ostream& o, rational r){
3    o<< r.n << "/" << r.d;
4    return o;
5  }
6
7  int main(){
8    rational r = {3,2};
9    cout<<r<<r<<endl;
10 }
```

Operatorüberladung auf codeboard Operatorpräzedenz

Const reference



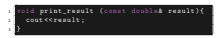
Const reference

- const vor einer Referenz heisst, dass das referenzierte Datum nicht veraendert werden kann.

```
1 int a = 5;
2 int& b = a;
3 const int& c = a;
4
5 c++; // runtime error
6 b++; // a is now 6
7 a++; // a is now 7
```

- result ist somit eine Referenz auf eine const double
- Lesen wie bei Pointern von rechts nach links
- So wird result ganz sicher nicht veraendert!

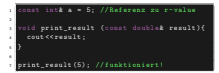
Const reference



Const reference

```
1 void print_result (const double& result){
2   cout<<result;
3 }
```

Const reference



Const reference

- Da c++ konstante Referenzen zu r-values erlaubt, ist es moeglich print_result mit Literalen aufzurufen. So schlaegt man zwei Fliegen mit einer Klappe. Erstens wird beim Aufruf ueber eine Variable keine Kopie erzeugt dank der Referenz und weiter kann die Funktion trotzdem noch auf r-values angewandt werden, was mit nicht const Referenzen nicht moeglich waere.

```
1 const int& a = 5; //Referenz zu r-value
2
3 void print_result (const double& result){
4   cout<<result;
5 }
6
7 print_result(5); //funktioniert!
```