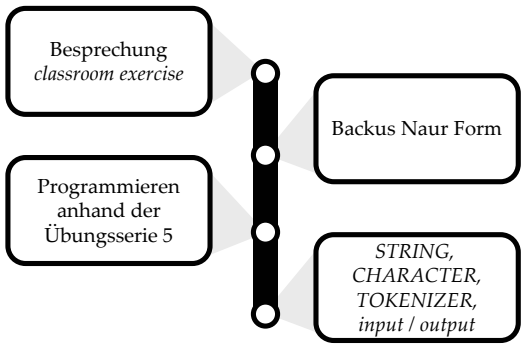
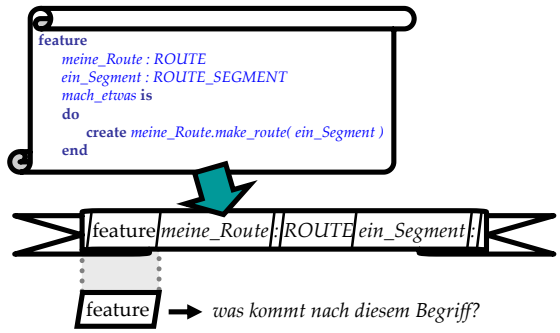
Montag, 24. November



Kompilieren



Backus-Naur-Form

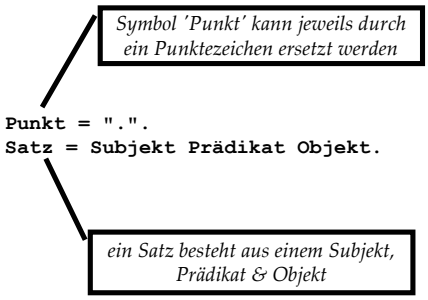
John Backus

Peter Naur

Backus und Nauer haben sich beim Erstellen von Compiler mit einer Notation für Grammatiken beschäftigt

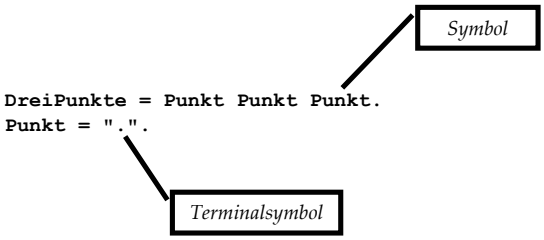
Syntaxregeln

Symbol = zu ersetzen durch.



Symbole & Terminalsymbole

- *Symbole sind Verweise auf Syntaxregeln*
- *Terminalsymbole sind abschliessende Symbole, geschrieben in Anführungs- und Schlusszeichen*



Etwas auf die Grammatik prüfen...

mit den Syntaxregeln lassen sich Sprachen definieren, so dass man überprüfen kann, ob etwas zu dieser Sprache gehört

Grammatik in BNF:

```
DreiPunkte = Punkt Punkt Punkt.
Punkt = ".".
```

zu überprüfendes Exemplar:

1. ist das ein **`DreiPunkte`**? nein
2. ist das ein Punkt? nein
3. ist also nicht, was wir mit der Grammatik definiert haben!

Mit den Terminalsymbolen kommen wir zu einem Ende

Terminalsymbole lassen sich nicht mehr weiter ersetzen – unsere Ersetzungen terminieren also irgendwann:

Grammatik in BNF:

```
DreiPunkte = Punkt Punkt Punkt.
Punkt = ".".
```

lässt sich nicht weiter ersetzen

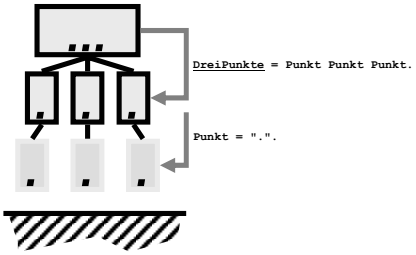
zu überprüfendes Exemplar:

1. ist das ein **`DreiPunkte`**? ja
2. **`DreiPunkte`** besteht aus **`Punkt Punkt Punkt`**
3. lässt sich das also weiter ersetzen? ja, mit **`4.Punkt`**, und das drei mal - fertig
5. es handelt sich dabei also um diese Sprache

als Baum

Terminalsymbole sind Blätter im Baum, Symbole sind innere Knoten und die erste Syntaxregel ist die Wurzel:

```
DreiPunkte = Punkt Punkt Punkt.
Punkt = ".".
```



es kann nichts mehr weiter ersetzt werden

Flexibilität

Bisher nur Aneinanderreihen (Konkatenation) von Symbolen – ziemlich einschränkend, darum

Alternative *– eines der angegebenen Dinge*

```
Vokal = "A" | "E" | "I" | "O" | "U".
```

"A" oder "E" oder "I" oder "O" oder "U"

Option *– entweder nehmen, oder nichts*

```
Flucht = "Ich renne" ["schnell"] "weg".
```

"Ich renne weg" oder "Ich renne schnell weg"

Repetitionen

* ***Repetition*** *– entweder nie, einmal, oder bis unendlich oft*

```
HeutigeTermine = { Zeit Termin ", " }*.
```

"" oder "15:00 Aufstehen, " oder "15:00 Aufstehen, 17:00 Essen"

\+ ***Repetition*** *– mindestens einmal, oder bis unendlich oft*

```
Testatliste = { Vorname " " Name "; " }+.
```

"Gianni Frey; " oder "Gianni Frey; Daniel Milani; "

Gruppierung

Die Klammern kann man in der BNF analog zur Mathematik verwenden

```
OhneKlammern = "1" | "2" | "3".
```

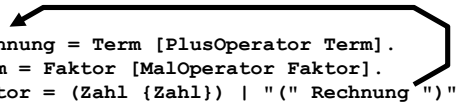
"1" oder "2" oder "3"

```
MitKlammern = ( "1" | "2" ) "3".
```

"13" oder "23"

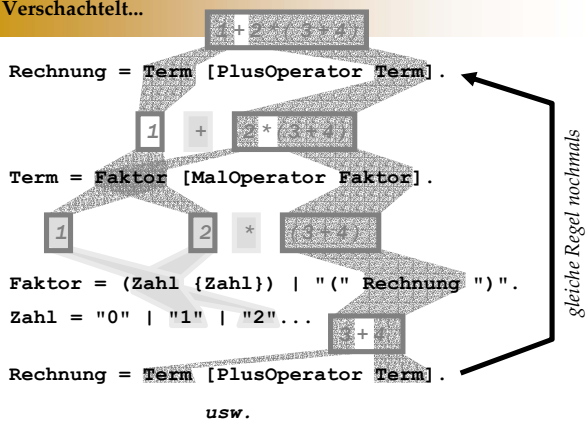
Rekursion

wenn in einer Auflistung solcher Syntaxregeln auf der rechten Seite Symbole, die weiter oben stehen, vorkommen:



```
Rechnung = Term [PlusOperator Term].
Term = Faktor [MalOperator Faktor].
Faktor = (Zahl {Zahl}) | "(" Rechnung ")".
Zahl = "0" | "1" | "2" | "3" | "4" |
 "5" | "6" | "7" | "8" | "9".
PlusOperator = "+".
MalOperator = "*".
```

Verschachtelt...



Programmieren

bisher:

1. Klassen definieren
2. Contracts spezifizieren

und jetzt:

3. Features sinnvoll auffüllen

dazu brauchen wir:

1. eine informelle Aufgabe (z.B. in Form der Aufgabenstellung)
2. daraus folgt eine formelle Aufgabe
3. und daraus folgt der Quellcode

1 Affenlatein

Folgende Regeln gelten:

- *Wenn ein wort mit einem oder mehreren Konsonanten beginnt, wird dieser Teil hinten angehängt und zusätzlich noch US angefügt*
  latein > l-atein > atein-l > atein-l-us > ateinlus
- *wenn das Wort mit einem Vokal beginnt, wird bus angefügt*
  affe > affe-bus > affebus
- *ein* **u** *nach einem* **q** *wird als Konsonant angesehen*
- *ein* **y** *gefolgt von einem Konsonanten wird als Vokal angesehen*
- *Worte, die kürzer als drei Zeichen sind, bleiben unverändert*

Schreibe einen Übersetzer, der alle eingegebenen Worte auf Affenlatein übersetzt!

Informell

erstelle informelle Beschreibung der Regeln:

- *entweder ist es ein ein-Buchstaben-Wort*
- *oder ein zwei-Buchstaben-Wort*
- *oder*
- *oder*

Formal

erstelle formelle Beschreibung der Regeln – also in BNF:

Wort = Buchstabe | Zweibuchstabenwort | Vokaluswort | Ybuswort .
Ybuswort = ...

der Übersetzer

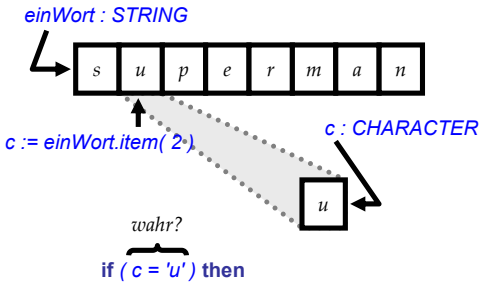
in einer neuen Klasse ein Feature, das übersetzt:

```
class TRANSLATOR

feature translate_to_affenlatein( s : STRING ) : STRING is
   require
        s_exists: s /= void
        s_is_valid: is_valid_input( s )
   local
        count: INTEGER
        tmp : STRING
   do
        -- ...hier musst du das ganze wort übersetzen...
   end
...
```

Zeichen & Zeichenketten

in Eiffel sind Worte Objekte der Zeichenkettenklasse (*STRING*)
eine Zeichenkette besteht aus Zeichen (*CHARACTER*)



Hilfsmittel für den Übersetzer

in der gleichen Klasse programmiert ihr noch Features,
welche die Übersetzung unterstützen:

```
...
feature is_valid_input( s : STRING ) : BOOLEAN is

feature is_vowel( c : CHARACTER ) : BOOLEAN is

feature is_consonant( c : CHARACTER ) : BOOLEAN is

feature is_treated_as_consonant( c : CHARACTER
                                 ) : BOOLEAN is

feature count_consonants_at_front( c : STRING
                                   ) : INTEGER is
...
```

Am Bildschirm zeigen & über Tastatur Eingetipptes lesen

das Zauberwort ***io.*** – *Input/Output*

io.put_string("Welcome to affenlatein translator!%N")

```
Welcome to affenlatein translator!
_
```

io.read_line -- liest die Zeile, die noch eingegeben wird, ein

```
Welcome to affenlatein translator!
gugus_
```

sobald man die Eingabetaste drückt, geht das Programm weiter

io.put_string(io.last_string) -- last_string enthält letzte Zeile

```
gugus
```

mehrere Worte

bisher nur jeweils ein Wort – jetzt wollen wir eine Reihe von Worten übersetzen...

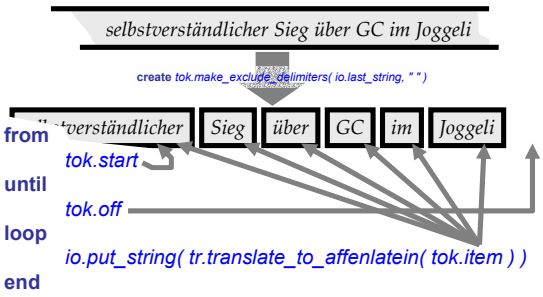
```
class TRANLATOR_APPLICATION
creation make
feature
   make is
   local
      tr : TRANSLATOR
      tok: STRING_TOKENIZER
   do
      create tr
      create tok.make_exclude_delimiters( io.last_string, " " )
      -- ... und so weiter ...
   end
end
```

unsere Klasse von vorher

macht Tokens aus einem Satz mit mehreren Worten

Tokenizer

der Stringtokenizer macht aus einem langen String mit mehreren Worten eine Liste mit Tokens (hier Worte)



2 Boolean Syntax

1. überlegen, was alles erlaubt ist bei logischen Ausdrücken
2. BNF aufschreiben

Boolean Ausdrücke bestehen immer aus **`True, False`** oder Identifiers mit irgendwelchen **`Not, And, Or, Implies, Or else,`** und **`and then`** darum herum

was sonst noch?

```
X or else Y
A = B
( A implies B ) = ( C or not ( A and then F ) )
```

3 Traffic XML

DTD ist ähnlich der BNF – anschauen und vergleichen

Map = "<map " Map_Attributes ">" Map_Elements "</map>" .
Map_Attributes = ["name=" Name] .
Map_Elements = ...