FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Compilerbau

WS 14/15

Heinz Faßbender
Raum 148
Tel. 0241/6009 51913
Email: fassbender@fh-aachen.de
Sprechstunde: nach Vereinbarung über Email

Organisatorisches

- **Vorlesungen und Übungen:**
  Di. 10.15 – 11.45 Raum G111
  Di. 12.15 – 13.45 Raum G111

- **Praktikum:**
  2 Stunden vor und nach der Vorlesung Raum G111
  Termine: 14.10., 04.11., 18.11., 02.12., 16.12., 13.01.

- **Sonstiges:**
  - im Download-Ordner
  - Regelmäßiges Vorrechnen von Übungsaufgaben verpfl.
  - Folien spätestens nach Vorlesung veröffentlicht
  - Übungsblatt nach Vorlesung
  - Praktikumsaufgabe normalerweise 1,5 Wochen vor Testat

- **Prüfung:**
  - mündlich am Anfang der Prüfungsperiode

Ablauf Praktikum

Versuch	Thema
1	Einführung JavaCC Parsen von arithmetischen Ausdrücken
2	Transformation Infix -> Postfix
3	Parsen von Mini-Java Symboltabelle
4	Übersetzung von arithmetischen Ausdrücken
5	Übersetzung von Mini-Java
6	Übersetzung von Prozeduren und Funktionen

Literaturhinweise

- Aho/Sethi/Ullman: *"Compilers – Principles, Techniques, Tools"*, Addison-Wesley, auch in Deutsch
- Waite/Goos: *"Compiler Construction"*, Springer, 1985
- Wilhelm/Maurer: *"Übersetzerbau: Theorie, Konstruktion, Generierung"*, Springer-Lehrbuch, 2. überarb. u. erw. Aufl., 1997
- Wirth: *"Grundlagen und Techniken des Compilerbaus"*, 2. bearbeitete Auflage, Oldenburg Verlag, 2008
- Schöning: *"Theoretische Informatik – kurzgefasst"*, Spektrum akademischer Verlag, 2001

Ziele

- Entwicklung eines Compilers **Mini-Java -> M32-Assembler**
- Compilererzeugendes Tool **JavaCC** kennenlernen
- **Phasen eines Compilers & Technologien** kennenlernen:
  - lexikalische Analyse — Scanner — reguläre Ausdrücke
  - Syntaxanalyse — Parser — EBNF
  - semantische Analyse — kaum — Attributgrammatiken
  - Codeerzeugung — — rekursive Funktionen
  - Codeoptimierung — — Datenflussanalyse, ...
- Anwendung der zugrunde liegenden **Theorie**
- Probleme beim **Entwurf von Programmiersprachen**
- **Äquvialenz** höhere Programmiersprache und Assembler
- **Komfort** höherer Programmiersprachen

Vorgehen

- kurzer Überblick über einzelne Compilerphasen
- Vorstellung der Sprache **Mini-Java**
- dabei: Wiederholung der zugrunde liegenden Konzepte
  - reguläre Ausdrücke
  - EBNF, CFG
  - Syntaxdiagramme
  - M32-Assembler
- Einführung in Compiler-erzeugendes Tool **JavaCC**
- theoretische Grundlagen der Compilerphasen:
  - Wie arbeitet JavaCC?
  - Wie Compiler von Hand schreiben

Voraussetzungen

Folgende Kenntnisse werden vorausgesetzt und nur kurz wiederholt:

- **Theoretische Informatik**

  Ahnung von Automatentheorie und formale Sprachen

- **ARBK**

  Ahnung vom M32-Assembler

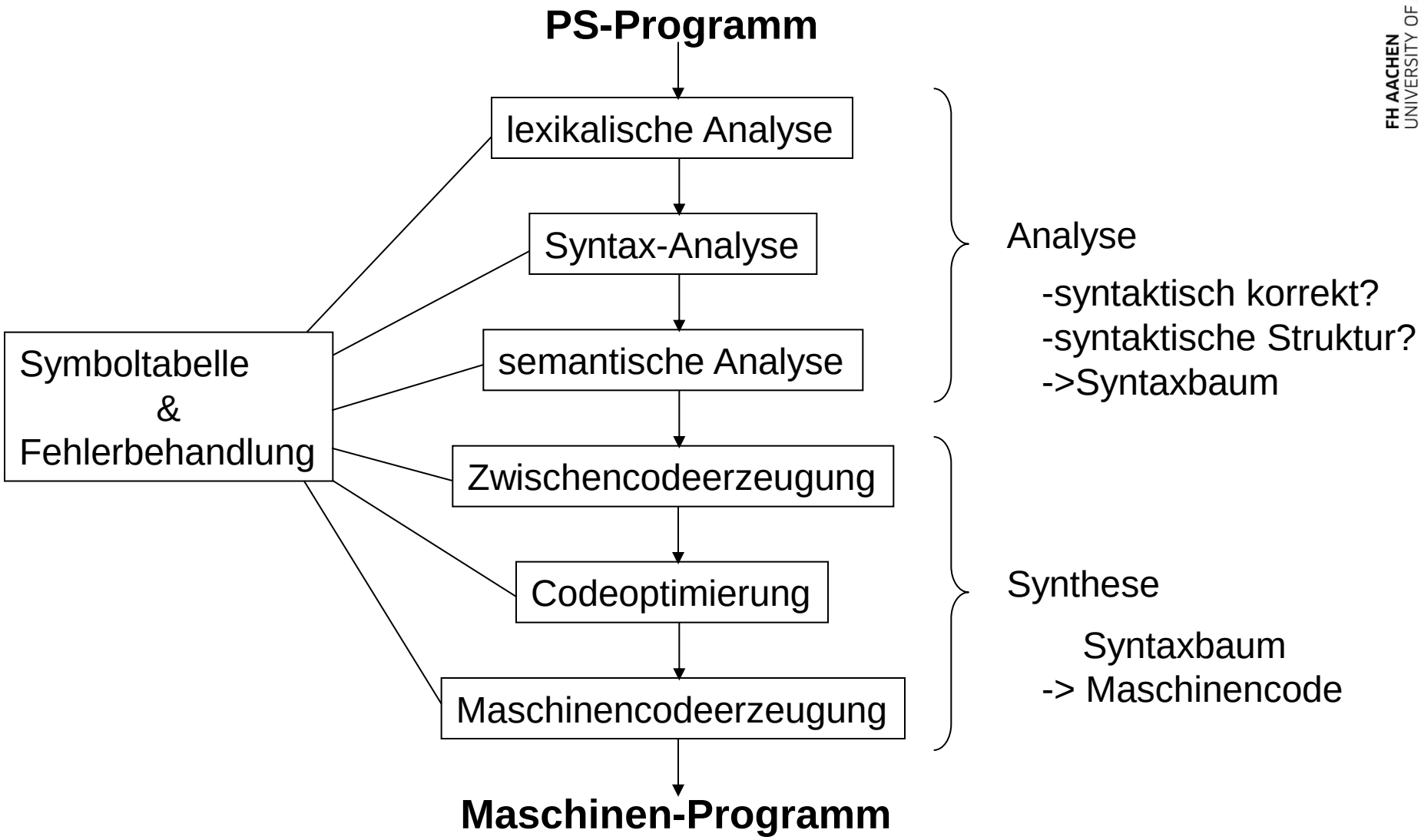
- **OOS**

  Ahnung von Java

Übersetzer

- **Ziel**
  Programme in höheren Programmiersprachen auf Rechnern ausführbar zu machen
- **Typen**
  - **Interpreter**
    jede Anweisung wird unabhängig analysiert, übersetzt und ausgeführt
  - **Compiler**
    übersetzt PS-Programm in Assemblerprogramm
  - **Java-Ansatz**
    mit Compiler PS-Programm in Bytecode übersetzen
    Bytecode durch JVM interpretieren
  - **hier: Compiler**

Phasen eines Compilers



lexikalische Analyse (kurz gefasst)

- **Scanner**
  - Programmteil, der lexikalische Analyse durchführt.
  - Unterprogramm des Parsers (Syntaxanalyse)

- **Warum separat?**
  - Parser: `bez1 = 354*bez2;` -> Zuweisung
    dabei: Name des Bezeichners und Zahlwerte egal
    Scanner liefert Zerlegung des Quelltextes in für Parser relevante Einheiten **(Lexeme)**: `IDE"="NUM"*"IDE";"`
  - einfache Struktur der Lexeme -> Erkennung effizient
  - Verbesserung der Portabilität
    Abhängigkeit vom Eingabealphabet auf Scanner beschränkt
  - Softwaretools für automatische Konstruktion von Scannern: **lex** & in **javacc** separater Teil

Lexemklassen (in Mini-Java)

- **Bezeichner** `IDENT`
  beliebig lange Folge von Buchstaben und Ziffern, die mit Kleinbuchstaben beginnt und kein Schlüsselwort ist.
- **Zahlen** `NUMBER`
  beliebig lange Folge von Ziffern, die nicht mit 0 beginnen, oder 0.
- **Schlüsselwörter**
  `final, int, if, else, while, print`
- **Einfache Symbole**
  `= ; { } ( ) < > , * / + -`
- **Zusammengesetze Symbole**
  `!= == <= >= // mit <,> als` **`compOP`** `bez.`

maschinenlesbare Beschreibung der Lexemklassen ?

Reguläre Ausdrücke (in javacc-Syntax)

- reichen zur Beschreibung der Lexemklassen in Compiler-erzeugenden Systemen (lex, javacc, ...), da Typ3-Sprachen

- **einige Konstrukte:**

  "X" -Festgelegte Zeichenkette.
  X kann eine beliebige Zeichenkette (String) sein.

  | -Alternative.

  (...)* -Inhalt der Klammern kommt nicht oder beliebig oft vor.

  (...)? -Inhalt der Klammern ist optional.

  (...)+ -Inhalt der Klammern kommt mindestens einmal vor.

  [...] -Inhalt der Klammern repräsentiert eine Menge von erlaubten Zeichen-(bereichen), durch Kommata voneinander getrennt.

  ~[...] -Inhalt der Klammern repräsentiert eine Menge von nicht erlaubten Zeichen(bereichen).

  (X){n} -Wiederhole Ausdruck X n-mal.

RA für `IDENT`?

Tokendefinition in javacc

- **Token:** Bezeichnung einer Lexemklasse
- in javacc-Konfigurationsdatei extra Abschnitt zur Definition von Token:

```
TOKEN:
  {
      < INT : "int" >          //Schlüsselwort
    |                          //könnte man auch weg lassen
      < FINAL : "final" >      //& später "int" einsetzen
    |
      < IDENT : ["a"-"z"] (["a"-"z","A"-"Z","0"-"9"])* >
  }
```

- **für einfache und zusammengesetzte Zeichen**
  üblicherweise keine Tokendefinition, sondern `"!="` oder `COMPOP`
- **für Mini-Java Zahlen** Praktikum Versuch 1
- **für Floating Point Zahlen** Übung

überlesene Symbole

- **Leerzeichen, Zeilenende, ...**
  dienen zur Trennung der Lexeme
  werden nicht an Parser weitergegeben, sondern überlesen
- **Kommentare**
  sollen zum Codeverständnis beitragen
  werden nicht an Parser weitergegeben, sondern überlesen
- **Pragmas**
  Compilerinformationen
  Optimierungen
- **Makros**
  werden von Präprozessor (in **C**) bearbeitet und transformiert

überlesene Symbole in javacc

- **Leerzeichen, Zeilenende, ...**
  in javacc-Konfigurationsdatei extra Abschnitt zur Definition von zu überlesenden Zeichen (üblicherweise vor TOKEN):

```
SKIP:
{
          " "           //Blank
    |     "\r"          //Wagenrücklauf (return)
    |     "\n"          //Neue Zeile (new line)
    |     "\t"          //Tabulator
}
```

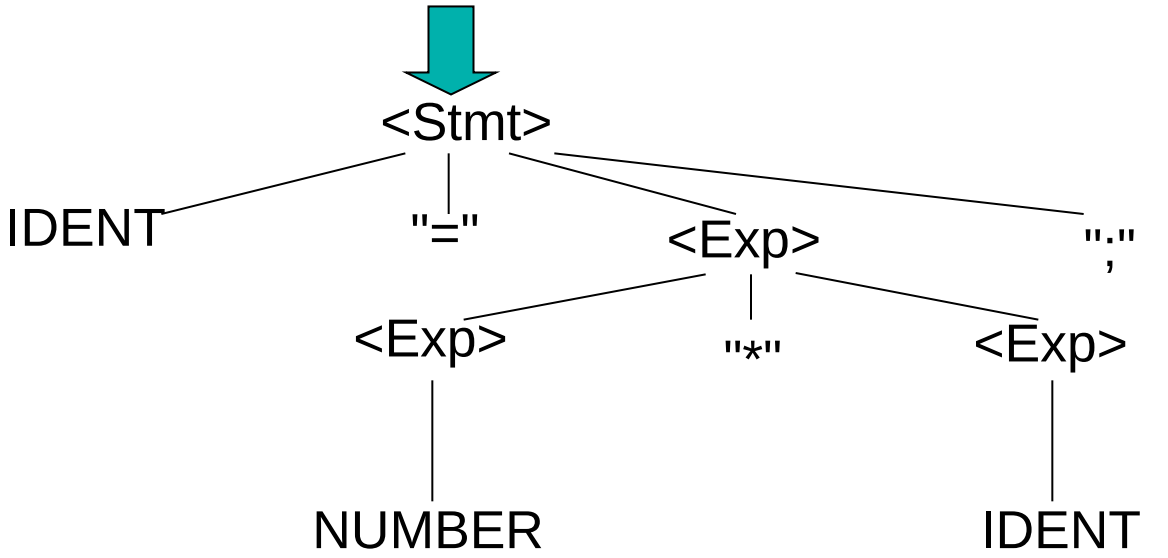
- **Kommentare**
  Übungsaufgabe

Syntaxanalyse (kurz gefasst)

- **Parser**
  - Programmteil, der Syntaxanalyse durchführt.
  - Eingabe: Tokenfolge von Scanner
  - Ausgabe: Syntaxbaum, falls Syntax o.k.
    meist in linearer Präfix-Darstellung

Beispiel

```
IDENT "=" NUMBER "*" IDENT ";"
```



1234

1243

Regeln in CFG?

Syntaxbeschreibungen

- **kontextfreie Grammatiken**
  da Syntax Typ2-Sprachen

  Bsp.: G = (V,Σ,S,R) mit

  - **Variablenmenge** V = {<Stmt>,<Exp>}
    (Nichtterminalsymbole)
  - **Terminalmenge** Σ = {IDENT, NUMBER, "=", ";", "*"}
  - **Startsymbol** S = <Stmt>
  - **Regelmenge** R =

```
{<Stmt>  ->   IDENT "=" <Exp> ";"  (1)
 <Exp>   ->   <Exp> "*" <Exp>      (2)
 <Exp>   ->   NUMBER               (3)
 <Exp>   ->   IDENT        }       (4)
```

- 1234 : Linksableitung
- 1243 : Rechtsableitung

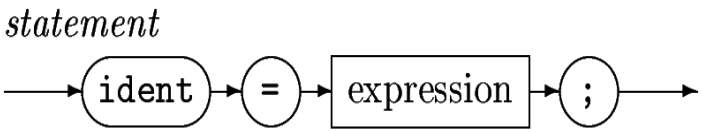
Syntaxbeschreibungen 2

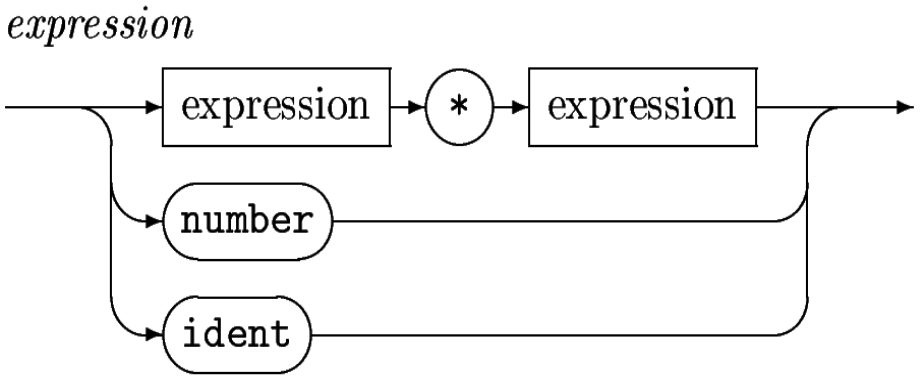
Extended Backus Naur Form EBNF

erlaube auf rechten Regelseiten einer kontextfreien Grammatik reguläre Ausdrücke

Bsp.: Ersetze in Regel 1 IDENT durch RA aus Tokendef.
<Stmt> -> ["a"-"z"] (["a"-"z","A"-"Z","0"-"9"])* "=" <Exp> ";"
Vorteil: Reduktion der Variablen

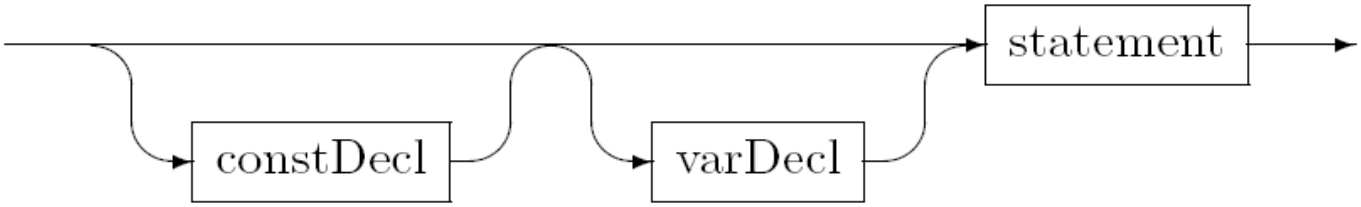
Syntaxdiagramme:



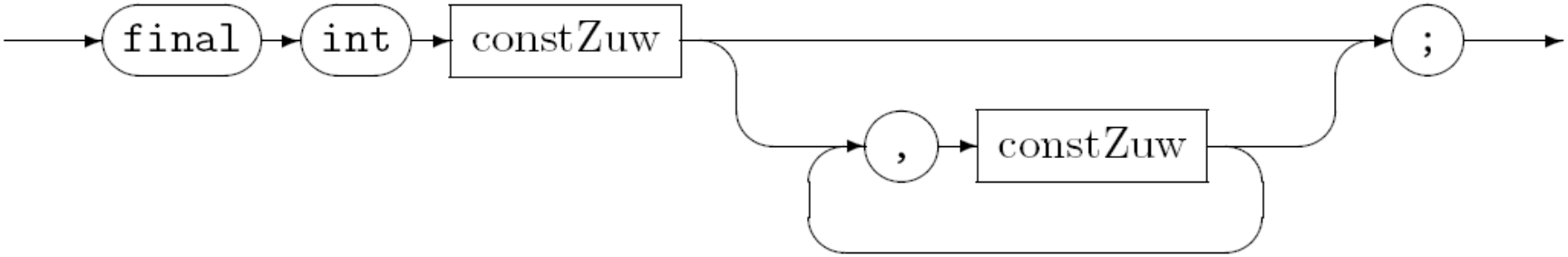


Syntax von Mini-Java

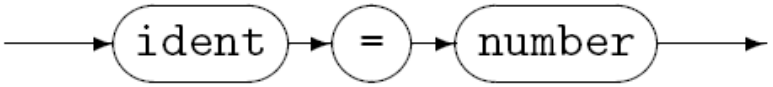
program



constDecl

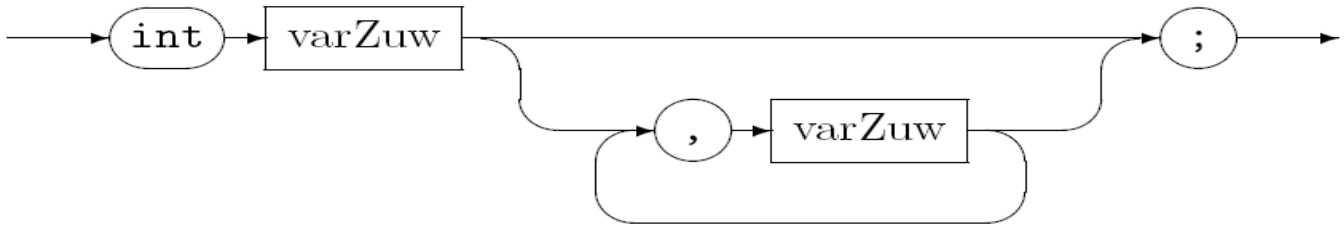


constZuw

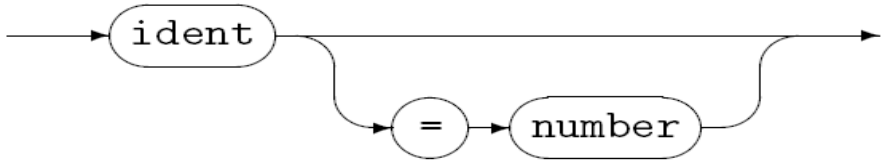


Syntax von Mini-Java 2

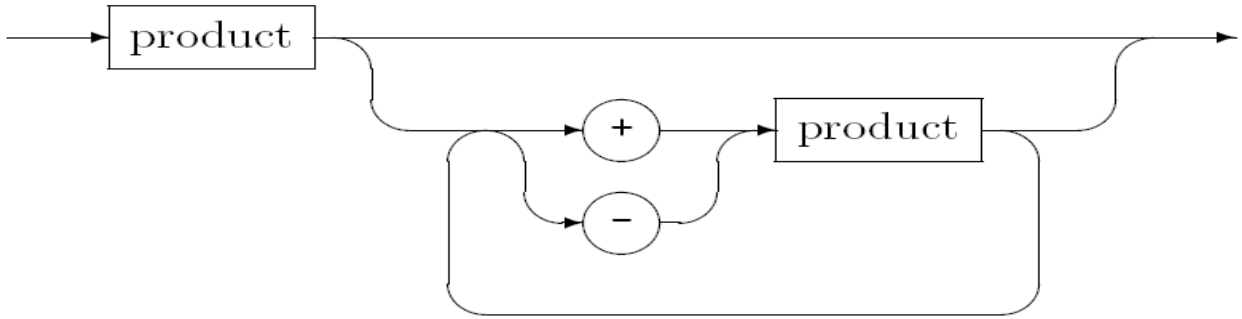
varDecl



varZuw

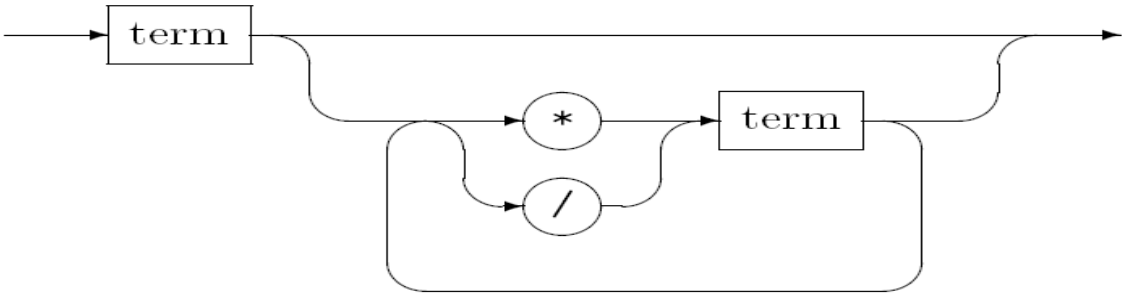


expression

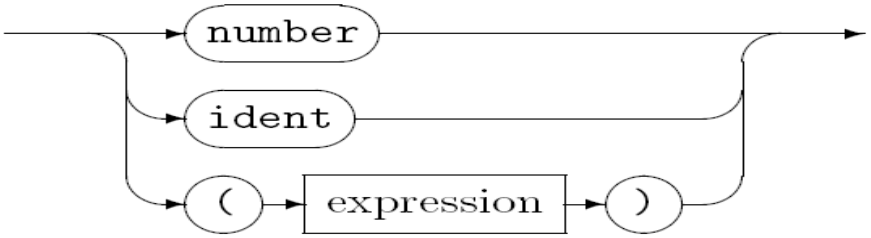


Syntax von Mini-Java 3

product

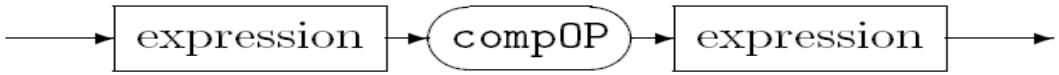


term

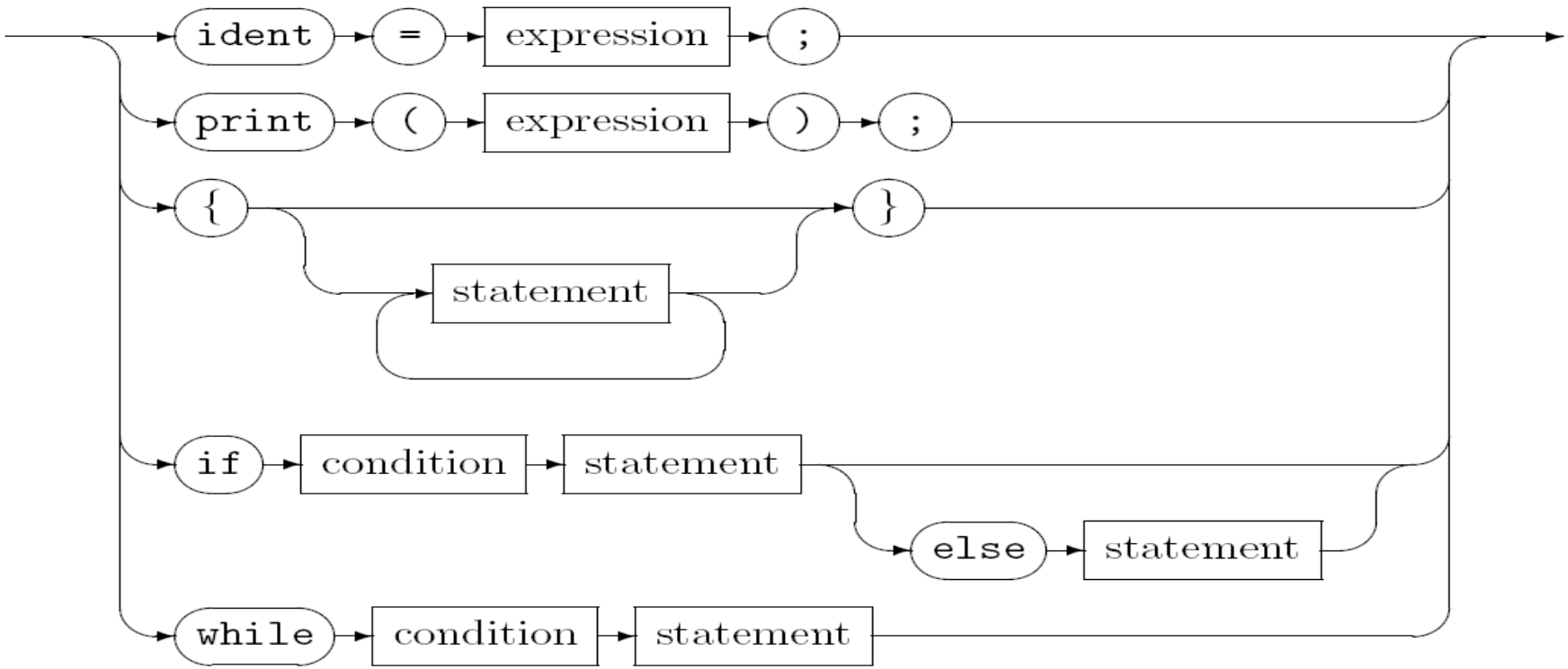


Syntax von Mini-Java 4

condition



statement



Syntaxdefinition in javacc

- in javacc-Konfigurationsdatei Syntaxdefinition in EBNF-ähnlicher Form: Startsymbol wird in main-Programm aufgerufen

```
void stmt():              //pro Variable eine Methode
{}                           //lokale Variablen
{                            //rechte Regelseiten (1)
 <IDENT> "=" exp() ";"    //Token in '<' '>'
}
void exp():
{}
{
 exp() "*" exp()
 |<NUMBER>                  //(2)-(4) mit oder
 |<IDENT>
}
```

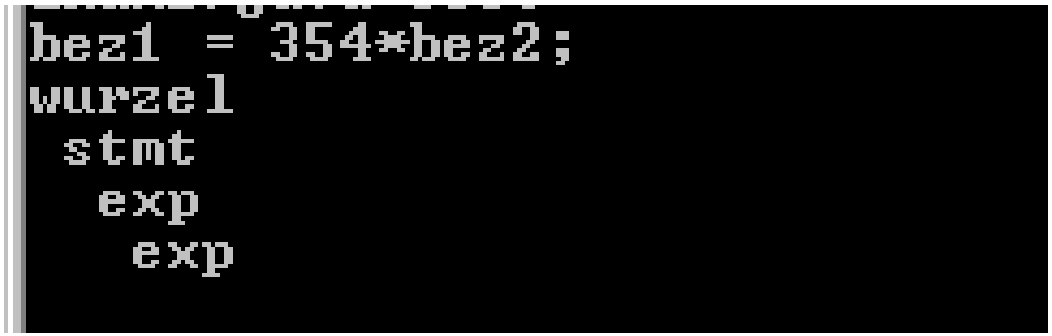
Regeln für Exp?

Syntaxbaum in javacc

- **jjtree** liefert **abstrakten Syntaxbaum** (Syntaxbaum ohne Blätter)
- ändere rechte Seiten in `exp()` wie folgt ab:

```
{
  "354" "*" exp()
 |<IDENT>
}
```

- füge neues Startsymbol `SimpleNode wurzel()` mit folgender Implementierung ein: `stmt() {return jjtThis;}`
- erhält bei Eingabe von `bez1 = 354*bez2; :`



statische Semantik von Mini-Java

- **Identifier müssen paarweise verschieden sein**
- **Deklarierte Variablen ohne Definition erhalten den Wert 0**
- **Bezeichner, die verwendet werden, müssen vorher deklariert sein.**
- **Bei Zuweisung links Variablenbezeichner**
- **Passt jede Argumentliste (Funktionsaufruf) zu Parameterliste in Funktionsdefinition**

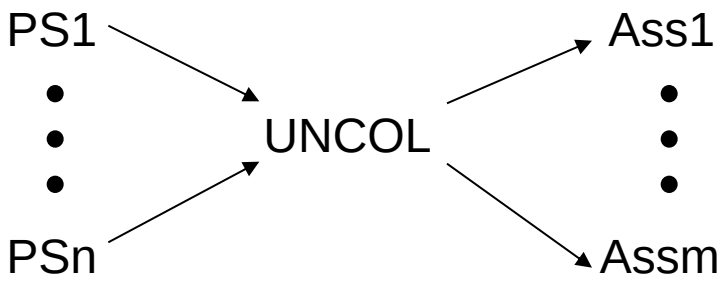
- **allgemeine Anmerkung zur Syntax**
  - compOP steht für < > <= >= == !=
  - in Übung Syntaxdiagramm ohne compOP

semantische Analyse (kurz gefasst)

- **Analyse** der Eigenschaften (**statische Semantik**), die nicht in Syntaxanalyse überprüft werden können:
  - kontextabhängige Eigenschaften (Syntax kontextfrei) z.B.:
    - Ist Bezeichner schon deklariert?
    - Typprüfungen
    - Umgebung von Blöcken und Prozeduren
    - siehe Folie "statische Semantik von Mini-Java"
- Lösung: **Attributgrammatiken**
- werden in Vorlesung nur knapp darauf eingehen
- unumgängliche Probleme erst zur Laufzeit abfangen, obwohl schon zur Übersetzungszeit möglich
  aber: hier zu aufwändig, darum pragmatische Lösung

Übersetzung in Zwischencode

- meist: nicht direkte Übersetzung in Assembler, sondern
  Übersetzung in Zwischencode **Z**: PS -> Z -> Assembler
- **Warum?**
  - Z abstrakter als Assembler -> Problembeseitigung aufteilen
  - bessere Portabilität des Compilers, siehe **Java Virtual Machine**
  - leichtere Codeoptimierung, da abstrakter
- Historisch:
  UNCOL (Universal Computer Oriented Language) (1960)



statt n*m Übersetzungen
nur n+m

Vorgehen in Vorlesung

- Beschreibung der Übersetzung als rekursive Funktionen, hierzu schrittweiser Aufbau von:
  - Programmiersprache (incl. Definition der Semantik **M**)
  - Zwischensprache (incl. Definition der Semantik **MZ**)
  - Übersetzung **trans: PS -> Z**
    mit M(P) = MZ(trans(P)), für alle Programme P aus PS

  in folgender Reihenfolge:
  - Ausdrücke auf Stackmaschine
  - Anweisungen durch Sprünge
  - Funktionen und Prozeduren durch Prozedurstack
  - Datenstrukturen

- **Bsp.:** Übersetzung eines Ausdrucks
  trans(7 + bez1) = LIT 7; LOAD(addr(bez1)); ADD;

Übersetzung mit javacc

- Übersetzung meist syntaxorientiert

-> Übersetzung wird meist verzahnt mit Parsing durchgeführt

- javacc bietet Möglichkeit beim Parsen Operationen auszuführen
  - Operationen können beliebiger Java-Code sein
  - Code muss in { } stehen, sonst Fehlermeldung

- insbesondere Codeerzeugung mit javacc:

```
String exp():                          //jetzt Rückgabewert
    ...
     |<NUMBER>
       {return token.image;} //Zahlwert zurückgeben
    ...
```

- Verwendung dieser erweiterten Möglichkeiten von javacc ab Praktikum Versuch 2 (Infix-Notation -> Postfix-Notation)

Codeoptimierung

- Elimination von nutzlosen Anweisungen:
  i gesetzt, aber nicht verwendet
- Elimination von redundanten Berechnungen:
  gleicher Funktionswert mehrmals berechnet
- Austausch von Befehlen:
  falls möglich und effizienter
- Optimierung von Sprüngen
- Datenflussanalyse:
  - nutzlose Pfade
  - nur ein Ast eines if's kann durchlaufen werden
  - While-Schleife keine Schleife, sondern nur Block

Vorgehen in Praktikum

- Umgang mit javacc kennenlernen
- reines Parsing für Ausdrücke
- Vorbereitung der Übersetzung durch
  - Transformation in Postfix
  - Verwendung von Symboltabellen
  - reines Parsing für Mini-Java
- Übersetzung im Gegensatz zur Vorlesung ohne Zwischencode, sondern direkt, aber unterteilt in Übersetzung von
  - Ausdrücken
  - Mini-Java
  - Mini-Java + Prozeduren & Funktionen

JavaCC-Konfigurationsdatei

```
options {
//Hier können noch Optionen stehen z.B. static
}
PARSER_BEGIN(ExprParse)      //Hier muss der Klassenname stehen
  public class ExprParse {
    public static void main (String args []) {
      ExprParse parser = new ExprParse(System.in);
      try {
        parser.ausdruck();              //Aufruf mit Startsymbol
        System.out.println("Ausdruck ok");
          } catch (ParseException e) {
              System.err.println(e);
          }
    }
  }
PARSER_END(ExprParse)
SKIP:   ...                     //siehe Folie
TOKEN: ...                      //siehe Folie
void ausdruck():                //Regeln für Startvariable und
...                               weitere Variablen siehe Folie
//überall kann bel. Java-Code hin,
//der immer an der Stelle ausgeführt, wo er steht
//weitere Infos in Beispielen zu JavaCC und Anleitung
```

Verwendung von javacc

- JavaCC-Konfigurationsdatei mit Name.jj abspeichern: `Test.jj`
- Erzeugung des Parsers mit `javacc Test.jj`
  liefert mehrere Dateien, u.a. `Test.java`
- Diese Datei übersetzen mit: `javac Test.java` und
- ausführen mit: `java Test`
- Standardeingabe und Standardausgabe ist die Konsole
- Verwendung von Piping um aus Dateien zu lesen oder in Dateien zu schreiben:

  ```
  (java Test < input.txt) > assembler.a32
  ```

- alternativ: PlugIn für Eclipse verwendbar

MiniJava-Compiler

- **fertiger Compiler für Mini-Java liegt auf Netz:**

  `MiniJavaCompiler.jar`

- **Verwendung zur Übersetzung von Programm in Datei**

  `MiniJavaCode.txt`

- **von Konsole aufrufen mit:**

  ```
  java -jar MiniJavaCompiler.jar < MiniJavaCode.txt
  ```

- **M32-Assembler (Swing-Version) liegt ebenfalls auf Netz:**

  `m32swing08.15.3.jar`