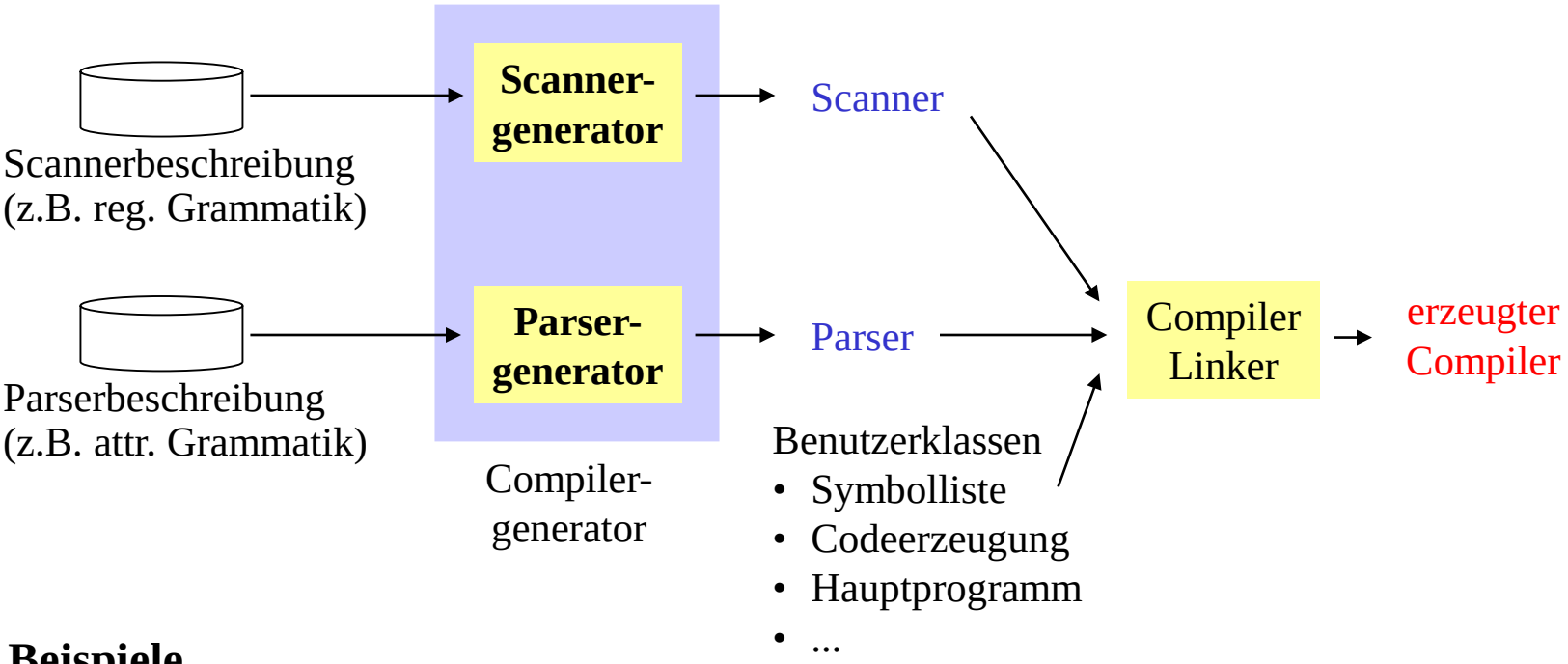
7. Compilergeneratoren

Hanspeter Mössenböck, Institut für Systemsoftware, JKU

Arbeitsweise von Compilergeneratoren

Erzeugen Compilerteile aus kompakter Spezifikation
(Compilerteile z.B. Scanner, Parser, Codegenerator, Baumoptimierer, ...)



Beispiele

Yacc	Parsergenerator für C und Java
Lex	Scannergenerator für C und Java
Coco/R	Scanner- u. Parsergenerator für Java, C#, C++, Delphi, Modula-2, Oberon, ...
...	

7. Compilergeneratoren

7.1 Überblick

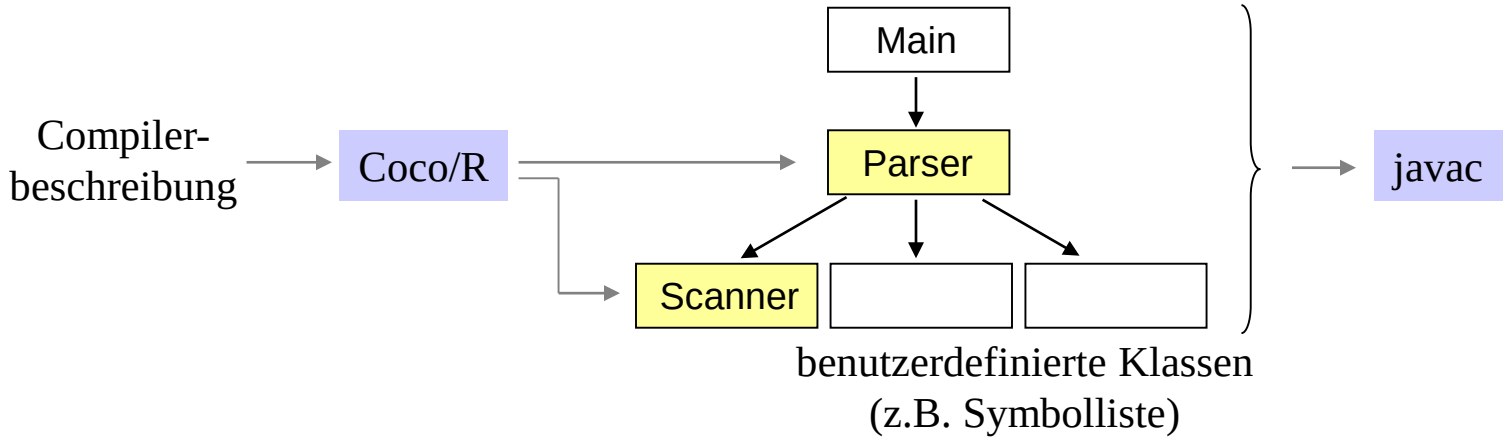
7.2 Coco/R

- Überblick
- Scannerbeschreibung
- Parserbeschreibung
- Fehlerbehandlung
- LL(1)-Konflikte

7.3 Beispiele

Coco/R - Compiler Compiler / Recursive Descent

Erzeugt aus einer ATG einen Scanner und einen Parser



Scanner	DFA
Parser	Rekursiver Abstieg
Herkunft	1980 an der JKU entwickelt
Heutige Versionen	für Java, C#, C++, VB.NET, Delphi, Modula-2, Visual Basic, Oberon, ...
Open Source	http://ssw.jku.at/Coco/

Beispiel: Compiler für Arithmetische Ausdrücke

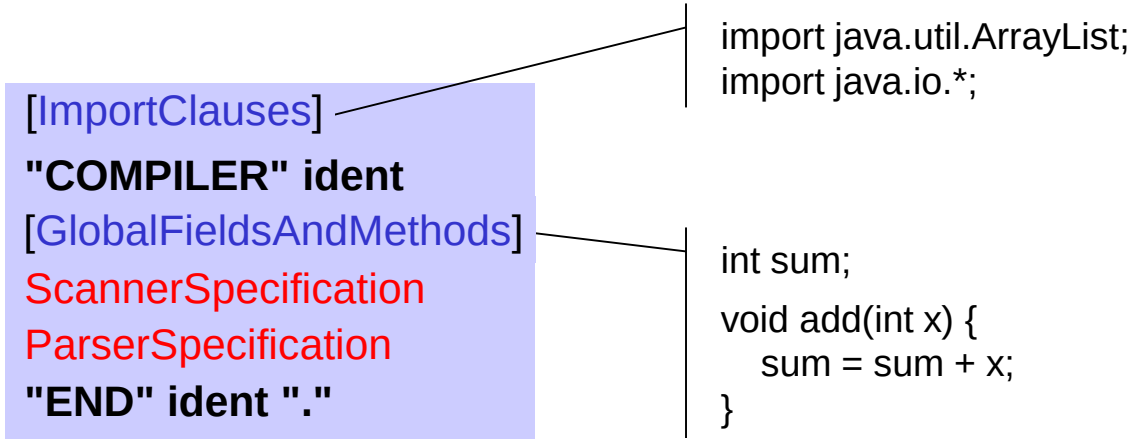

```
COMPILER Calc
CHARACTERS
   digit = '0' .. '9'.
TOKENS
   number = digit {digit}.
COMMENTS FROM "/*" TO "*/" NESTED
IGNORE '\t' + '\r' + '\n'
```

Scannerbeschreibung

```
PRODUCTIONS
   Calc                          (. int x; .)
   = "CALC" Expr<out x>          (. System.out.println(x); .) .

   Expr <out int x>              (. int y; .)
   =  Term<out x>
      { '+' Term<out y>          (. x = x + y; .)
      }.

   Term <out int x>              (. int y; .)
   =  Factor<out x>
      { '*' Factor<out y>        (. x = x * y; .)
      }.

   Factor <out int x>
   =  number                     (. x = Integer.parseInt(t.val); .)
   |  '(' Expr<out x> ')'.
END Calc.
```

Parserbeschreibung

Struktur einer Compilerbeschreibung



ident bezeichnet das Startsymbol der Grammatik (d.h. das oberste NTS)

7. Compilergeneratoren

7.1 Überblick

7.2 Coco/R

- Überblick
- Scannerbeschreibung
- Parserbeschreibung
- Fehlerbehandlung
- LL(1)-Konflikte

7.3 Beispiele

Struktur der Scannerbeschreibung


```
ScannerSpecification =
   ["IGNORECASE"]
   ["CHARACTERS" {SetDecl}]
   ["TOKENS" {TokenDecl}]
   ["PRAGMAS" {PragmaDecl}]
   {CommentDecl}
   {WhiteSpaceDecl}.
```

- Soll der erzeugte Compiler case-sensitiv sein?
- Welche Zeichen dürfen in Token vorkommen?
- Syntax der Token (Terminalklassen)
- Pragmas sind Token, die nicht Teil der Grammatik sind
- Beschreibung, wie Kommentare aufgebaut sind
- Welche Zeichen sollen ignoriert werden (z.B. \t, \n, \r)?

Definition der Zeichenmengen

Beispiele

```
CHARACTERS
   digit    = "0123456789".
   letter   = 'A' .. 'Z'.
   hexDigit = digit + "ABCDEF".
   eol      = '\n'.
   noDigit  = ANY - digit.
```

Menge aller Ziffern
Menge aller Großbuchstaben
Menge aller Hexadezimalziffern
das end-of-line-Zeichen
beliebiges Zeichen, das keine Ziffer ist

Folgende Escape-Zeichen können dabei verwendet werden

\\	backslash	\r	carriage return	\f	form feed
\'	apostrophe	\n	new line	\a	bell
\"	quote	\t	horizontal tab	\b	backspace
\0	null character	\v	vertical tab	\uxxxx	hex character value

Coco/R erlaubt Unicode (UTF-8)

Token-Deklarationen

Definieren die Syntax von Terminalklassen (z.B. ident, number, ...)
Literale wie "while" oder ">=" müssen nicht deklariert werden

Beispiele

```
TOKENS
   ident     =  letter {letter | digit | '_'}.
   number    =  digit {digit}
             |  "0x" hexDigit hexDigit hexDigit hexDigit.
   float     =  digit {digit} '.' digit {digit} ['E' ['+' | '-'] digit {digit}].
```

kein Problem, wenn Alternativen
mit dem gleichen Zeichen beginnen

- Rechte Seite muss eine reguläre EBNF-Regel sein
- Namen auf der rechten Seite bezeichnen Zeichenmengen

Pragmas

Spezielle Token (z.B. Compileroptionen, Präprozessor-Kommandos)

- können an beliebiger Stelle der Eingabe vorkommen
- sind nicht Teil der Grammatik
- müssen semantisch verarbeitet werden

Beispiel: $ABC

```
PRAGMAS
   option = '$' {letter}.  (.  for (char ch: la.val.toCharArray()) {
                                  if (ch == 'A') ...
                                  else if (ch == 'B') ...
                                  ...
                               } .)
```

immer, wenn in der Eingabe ein *option*-Token vorkommt, (z.B. $ABC) wird diese semantische Aktion ausgeführt

Typische Anwendungen

- Compileroptionen
- Präprozessor-Kommandos (z.B. #ifdef)
- Kommentarverarbeitung (z.B. javadoc)
- end-of-line-Verarbeitung

Kommentare

Werden in einem speziellen Abschnitt beschrieben

- weil sie der Parser (im Gegensatz zu anderen Token) ignorieren muss
- weil geschachtelte Kommentare nicht mit regulären Grammatiken beschrieben werden können

Beispiele

```
COMMENTS FROM "/*" TO "*/" NESTED
COMMENTS FROM "//" TO "\r\n"
```

White Space und Case-Sensitivität

White space

```
IGNORE   '\t' + '\r' + '\n'
```

Zeichenmenge

Leerzeichen werden standardmäßig immer ignoriert

Case-Sensitivität

Erzeugte Compiler sind standardmäßig Case-sensitiv (d.h. Foo != foo)

Wenn man das nicht will, muss man `IGNORECASE` verwenden

```
COMPILER Sample
IGNORECASE
CHARACTERS
   hexDigit = digit + 'a'..'f'.
   ...
TOKENS
   number = "0x" hexDigit hexDigit hexDigit hexDigit.
   ...
PRODUCTIONS
   WhileStat = "while" '(' Expr ')' Stat.
   ...
END Sample.
```

erkennt

- 0x00ff, 0X00ff, 0X00FF als *number*
- while, While, WHILE als Schlüsselwort

Token-Werte (Namen, Strings) werden mit originaler Groß/Kleinschreibung an den Parser geliefert

Schnittstelle des erzeugten Scanners


```
public class Scanner {
   public Scanner (String fileName);
   public Scanner (InputStream s);
   public Token    Scan();
   public Token    Peek();
   public void     ResetPeek();
}
```

wichtigste Methode:
liefert bei jedem Aufruf ein Token

liest von der momentanen Scannerposition voraus, ohne Token aus dem Eingabestrom zu entfernen

setzt die Peek-Position auf die momentane Scannerposition zurück

```
public class Token {
   public int    kind; // token kind (i.e. token number)
   public String val;  // token value
   public int    pos;  // token position in the source text (starting at 0)
   public int    col;  // token column (starting at 1)
   public int    line; // token line (starting at 1)
}
```

7. Compilergeneratoren

7.1 Überblick

7.2 Coco/R

- Überblick
- Scannerbeschreibung
- Parserbeschreibung
- Fehlerbehandlung
- LL(1)-Konflikte

7.3 Beispiele

Produktionen

- Können in beliebiger Reihenfolge angegeben werden
- Für jedes NTS muss es genau 1 Produktion geben
- Es muss eine Produktion für das Startsymbol (den Grammatiknamen) geben

Beispiel

```
COMPILER Expr
   ...
PRODUCTIONS
   Expr      =  SimExpr [RelOp SimExpr].
   SimExpr   =  Term {AddOp Term}.
   Term      =  Factor {Mulop Factor}.
   Factor    =  ident | number | "-" Factor | "true" | "false".
   RelOp     =  "==" | "<" | ">".
   AddOp     =  "+" | "-".
   MulOp     =  "*" | "/".
END Expr.
```

Beliebige kontextfreie Grammatik in EBNF

Semantische Aktionen

Beliebiger Java-Code zwischen (. und .)

```
IdentList       (. int n; .)
=  ident        (. n = 1; .)
   { ',' ident  (. n++; .)
   }            (. System.out.println(n); .)
   .
```

lokale semantische Deklaration

semantische Aktion

Sem. Aktionen werden in den generierten Parser kopiert, ohne von Coco/R geprüft zu werden

Globale semantische Deklarationen

```
import java.io.*;
COMPILER Sample
   FileWriter w;
   void open(String path) {
      w = new FileWriter(path);
      ...
   }
...
PRODUCTIONS
   Sample = ...      (. open("in.txt"); .)
   ...
END Sample.
```

Import von Klassen aus anderen Paketen

globale semantische Deklarationen
(werden zu Feldern und Methoden des Parsers)

semantische Aktionen können auf globale Deklarationen und auf importierte Klassen zugreifen

Coco/R — Attribute

Terminalsymbole

- haben keine expliziten Attribute
- ihre Werte können in sem. Aktionen über folgende Parser-Variablen angesprochen werden

  Token **t**; das zuletzt erkannte Token

  Token **la**; das noch nicht erkannte Lookahead-Token

Beispiel

```
Factor = number    (. int x = Integer.parseInt(t.val); ... .)
```

```
class Token {
   int kind;        // token code
   String val;      // token value
   int pos;         // token position in the source text (starting at 0)
   int line;        // token line (starting at 1)
   int col;         // token column (starting at 1)
}
```

Nonterminalsymbole

- beliebig viele Eingangsattribute

  formale Attr.: `A <int x, char c> = ... .` aktuelle Attr.: `... A <y, 'a'> ...`

- höchstens ein Ausgangsattribut (muss das erste in der Attributliste sein)

  `B <out int x, int y> = ... .` `... B <out z, 3> ...`

Produktionen werden in Parsermethoden übersetzt

Produktion

```
Expr<out int n>          (. int n1; .)
=  Term<out n>
   {  '+'
      Term<out n1>       (. n = n + n1; .)
   }.
```

Erzeugte Parsermethode

```
int Expr() {
   int n;
   int n1;
   n = Term();
   while (la.kind == 3) {
      Get();
      n1 = Term();
      n = n + n1;
   }
   return n;
}
```

Attribute => Parameter oder Rückgabewerte
Semantische Aktionen => eingebetteter Code im Parser

Das Symbol ANY

Bedeutet alle Teminalsymbole, die nicht Alternative zu diesem ANY sind
(in der enthaltenden Produktion)

"Fuzzy Parsing"

Beispiele

Anzahl der Vorkommen von *int* zählen

```
Type
=  "int"          (. intCounter++; .)
|  ANY.
```

jedes Terminalsymbol außer "int"

Länge einer semantischen Aktion berechnen

```
SemAction<out int len>
=  "(."          (. int beg = t.pos + 2; .)
   { ANY }
   ".)"          (. len = t.pos - beg; .) .
```

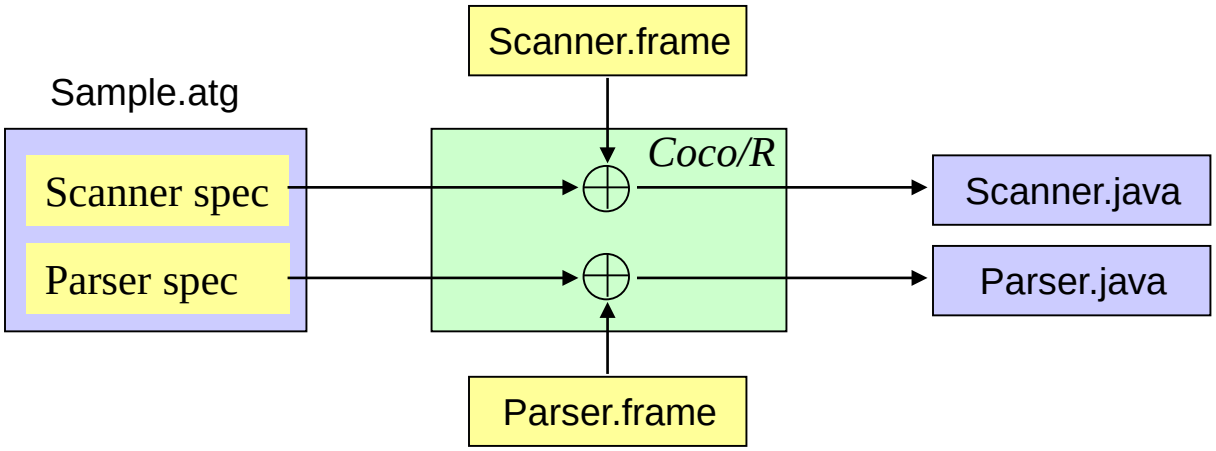
jedes Terminalsymbol außer ".)"

Anweisungen (Strichpunkte) in einem Block zählen

```
Block<out int stmts>    (. int n; .)
=  "{"                  (. stmts = 0; .)
   { ";"                (. stmts++; .)
   | Block<out n>       (. stmts += n; .)
   | ANY
   }
   "}".
```

alles, was nicht "{", "}" oder ";" ist

Frame-Dateien



Scanner.frame (Auszug)

```
public class Scanner {
   static final char EOL = '\n';
   static final int eofSym = 0;
-->declarations
   ...
   public Scanner (InputStream s) {
      buffer = new Buffer(s);
      Init();
   }
   void Init () {
      pos = -1; line = 1; …
-->initialization
   ...
}
```

- Coco/R fügt erzeugte Teile an Stellen ein, die mit "-->..." markiert sind
- Benutzer können Frame-Dateien editieren und dadurch Scanner und Parser ihren Bedürfnissen anpassen
- Frame-Dateien müssen im selben Verzeichnis sein wie die ATG-Datei

Schnittstelle des erzeugten Parsers


```
public class Parser {
   public Scanner scanner;   // the scanner of this parser
   public Errors  errors;    // the error message stream
   public Token t;           // most recently recognized token
   public Token la;          // lookahead token
   public Parser (Scanner scanner);
   public void Parse ();
   public void SemErr (String msg);
}
```

Aufruf des Parsers aus dem Hauptprogramm

```
public class MyCompiler {

   public static void main(String[] arg) {
      Scanner scanner = new Scanner(arg[0]);
      Parser parser = new Parser(scanner);
      parser.Parse();
      System.out.println(parser.errors.count + " errors detected");
   }
}
```

7. Compilergeneratoren

7.1 Überblick

7.2 Coco/R

- Überblick
- Scannerbeschreibung
- Parserbeschreibung
- Fehlerbehandlung
- LL(1)-Konflikte

7.3 Beispiele

Syntaxfehler-Behandlung

Syntaxfehlermeldungen werden automatisch erzeugt

Für fehlerhafte Terminalsymbole

Produktion	S = a b c.
Eingabe	a x c
Fehlermeldung	-- line ... col ...: b expected

Für fehlerhafte Alternativenlisten

Produktion	S = a (b | c | d) e.
Eingabe	a x e
Fehlermeldung	-- line ... col ...: invalid S

Fehlermeldungen können oft durch Umschreiben der Grammatik verbessert werden

Produktionen	S = a T e. T = b | c | d.
Eingabe	a x e
Fehlermeldung	-- line ... col ...: invalid T

Wiederaufsatz nach Syntaxfehlern

Benutzer muss Synchronisationspunkte angeben, wo Wiederaufsatz stattfinden soll

Synchronisationspunkte

```
Statement
= SYNC
   (   Designator "=" Expr SYNC ';'
   |   "if" '(' Expression ')' Statement ["else" Statement]
   |   "while" '(' Expression ')' Statement
   |   '{' {Statement} '}'
   |   ...
   ).
```

Was geschieht, wenn ein Fehler entdeckt wird?

- Parser meldet ihn
- setzt bis zum nächsten Synchronisationspunkt fort
- überliest Eingabesymbole bis er eines findet, das am Sychronisationspunkt erlaubt ist

```
while (la.kind is not accepted here) {
   la = scanner.Scan();
}
```

Was sind gute Synchronisationspunkte?

Stellen in der Grammatik, an denen besonders "sichere" Terminalsymbole erwartet werden

- Beginn von Statement: if, while, do, ...
- Beginn von Declaration: public, static, void, ...
- vor einem Strichpunkt

Semantikfehler-Behandlung

Muss in semantischen Aktionen codiert werden

```
Expr<out Type type>          (.  Type type1; .)
=  Term<out type>
   {   '+' Term<out type1>   (.  if (type != type1) SemErr("incompatible types"); .)
   } .
```

Methode *SemErr* im Parser

```
void SemErr (String msg) {
    ...
    errors.SemErr(t.line, t.col, msg);
    ...
}
```

Vorsicht

Nach Syntaxfehlern haben u.U. einige Variablen einen ungültigen Wert (wegen Wiederaufsatz)

Klasse Errors

Coco/R erzeugt eine Klasse zur Ausgabe von Fehlermeldungen

```
public class Errors {

   public int count = 0;                                  // number of errors detected
   public PrintStream errorStream = System.out;           // error message stream
   public String errMsgFormat = "-- line {0} col {1}: {2}";   // 0=line, 1=column, 2=text

   // called by the programmer (via Parser.SemErr) to report semantic errors
   public void SemErr (int line, int col, String msg) {
      printMsg(line, col, msg);
      count++;
   }

   // called automatically by the parser to report syntax errors
   public void SynErr (int line, int col, int n) {
      String msg;
      switch (n) {
         case 0: msg = "..."; break;
         case 1: msg = "..."; break;
         ...
      }
      printMsg(line, col, msg);
      count++;
   }
   ...
}
```

Syntaxfehlermeldungen (von Coco/R erzeugt)

7. Compilergeneratoren

7.1 Überblick

7.2 Coco/R

- Überblick
- Scannerbeschreibung
- Parserbeschreibung
- Fehlerbehandlung
- LL(1)-Konflikte

7.3 Beispiele

Coco/R findet LL(1)-Konflikte automatisch

Beispiel

```
...
PRODUCTIONS
   Sample     =  {Statement}.
   Statement  =  Qualident '=' number ';'
                 |  Call
                 |  "if" '(' ident ')' Statement ["else" Statement].
   Call       =  ident '(' ')' ';'.
   Qualident  =  [ident '.'] ident.
   ...
```

Coco/R erzeugt folgende Warnungen

```
>coco Sample.atg
Coco/R (Apr 19, 2022)
checking
   Sample deletable
   LL1 warning in Statement: ident is start of several alternatives
   LL1 warning in Statement: "else" is start & successor of deletable structure
   LL1 warning in Qualident: ident is start & successor of deletable structure
parser + scanner generated
0 errors detected
```

Konfliktlösung durch Vorausschau

```
A = ident (. x = 1; .) {',' ident (. x++; .) } ':'
  | ident (. Foo(); .) {',' ident (. Bar(); .) } ';'.
```

LL(1)-Konflikt

Auflösung

```
A = IF (followedByColon())
      ident (. x = 1; .) {',' ident (. x++; .) } ':'
  | ident (. Foo(); .) {',' ident (. Bar(); .) } ';'.
```

"Conflict Resolver"

Auflösungsmethode

```
boolean followedByColon() {
   Token x = la;
   while (x.kind == _ident || x.kind == _comma) {
      x = scanner.Peek();
   }
   return x.kind == _colon;
}
```

```
TOKENS
   ident = letter {letter | digit} .
   comma = ','.
   ...
```

```
static final int
   _ident = 17,
   _comma = 18,
   ...
```

Konfliktlösung durch sem. Information


```
Factor = '(' ident ')' Factor      /* type cast */
       | '(' Expr ')'              /* nested expression */
       | ident | number.
```

LL(1)-Konflikt

Auflösung

```
Factor = IF (isCast())
         '(' ident ')' Factor      /* type cast */
       | '(' Expr ')'              /* nested expression */
       | ident | number.
```

Auflösungsmethode

```
boolean isCast() {
   Token next = scanner.Peek();
   if (la.kind == _lpar && next.kind == _ident) {
      Obj obj = Tab.find(next.val);
      return obj != Tab.noObj && obj.kind == Obj.Type;
   } else return false;
}
```

liefert *true*, wenn nach dem '(' ein Typname kommt

7. Compilergeneratoren

7.1 Überblick

7.2 Coco/R

7.3 Beispiele

- Fragebogen-Generator
- Lesen eines Binärbaums
- Erzeugung abstrakter Syntaxbäume

Fragebogen-Generator

Eingabe: Domänenspezifische Sprache zur Beschreibung von Fragebögen

```
RADIO "How did you like this course?"
("very much", "much", "somewhat", "not so much", "not at all")
CHECKBOX "What is the field of your study?"
("Computer Science", "Mathematics", "Physics")
TEXTBOX "What should be improved?"
...
```

Ausgabe: HTML-Fragebogen

How did you like this course?
- ○ very much
- ○ much
- ○ somewhat
- ○ not so much
- ○ not at all

What is the field of your study?
- ☐ Computer Science
- ☐ Mathematics
- ☐ Physics

What should be improved?

Was ist zu tun?

1. Eingabesprache durch Grammatik beschreiben
2. Attribute für die Symbole definieren
3. Semantische Routinen definieren
4. ATG schreiben

Grammatik der Eingabe


```
QueryForm   = {Query}.
Query       = "RADIO" Caption Values
            | "CHECKBOX" Caption Values
            | "TEXTBOX" Caption.
Values      = '(' string {',' string} ')'.
Caption     = string.
```

```
RADIO "How did you like this course?"
("very much", "much", "somewhat",
 "not so much", "not at all")

CHECKBOX "What is the field of your study?"
("Computer Science", "Mathematics", "Physics")

TEXTBOX "What should be improved?"
```

Attribute

- Caption liefert einen String — Caption<out String s>
- Values liefert eine Liste von Strings — Values<out ArrayList list>

Semantische Routinen

- printHeader()
- printFooter()
- printRadio(caption, values)
- printCheckbox(caption, values)
- printTextbox(caption)

} in Klasse HtmlGenerator implementiert

Scannerbeschreibung


```
COMPILER QueryForm

CHARACTERS
   noQuote = ANY - "".
   tab = '\t'.
   cr = '\r'.
   lf = '\n'.

TOKENS
   string = "" {noQuote} "".

COMMENTS
   FROM "//" TO cr lf

IGNORE tab + cr + lf

...

END QueryForm.
```

Parserbeschreibung

```
import java.util.ArrayList;
COMPILER QueryForm
   HtmlGenerator html;
   ...
PRODUCTIONS
QueryForm =                          (. html.printHeader(); .)
   { Query }                         (. html.printFooter(); .) .
//-------------------------------------------------------------------------------------
Query                                (. String caption; ArrayList values; .)
=  "RADIO" Caption<out caption> Values<out values>
                                     (. html.printRadio(caption, values); .)
|  "CHECKBOX" Caption<out caption> Values<out values>
                                     (. html.printCheckbox(caption, values); .)
|  "TEXTBOX" Caption<out caption>
                                     (. html.printTextbox(caption); .) .
//-------------------------------------------------------------------------------------
Caption<out String s> = StringVal<out s>.
//-------------------------------------------------------------------------------------
Values<out ArrayList values>         (. String s; .)
=  '(' StringVal<out s>              (. values = new ArrayList(); values.add(s); .)
   { ',' StringVal<out s>            (. values.add(s); .)
   }
   ')'.
//-------------------------------------------------------------------------------------
StringVal<out String s>
=  string                            (. s = t.val.substring(1, t.val.length()-1); .) .
END QueryForm.
```

Klasse HtmlGenerator

```
import java.io.*;
import java.util.ArrayList;

class HtmlGenerator {
   PrintStream s;
   int itemNo = 0;

   public HtmlGenerator(String fileName) throws FileNotFoundException {
      s = new PrintStream(fileName);
   }

   public void printHeader() {
      s.println("<html>");
      s.println("<head><title>Query Form</title></head>");
      s.println("<body>");
      s.println("  <form>");
   }

   public void printFooter() {
      s.println("  </form>");
      s.println("</body>");
      s.println("</html>");
      s.close();
   }
   ...
```

Klasse HtmlGenerator *(Forts.)*

```
    public void printRadio(String caption, ArrayList values) {
        s.println(caption + "<br>");
        for (Object val: values) {
            s.print("<input type='radio' name='Q" + itemNo + "' ");
            s.print("value='" + val + "'>" + val + "<br>");
            s.println();
        }
        itemNo++; s.println("<br>");
    }

    public void printCheckbox(String caption, ArrayList values) {
        s.println(caption + "<br>");
        for (Object val: values) {
            s.print("<input type='checkbox' name='Q" + itemNo + "' ");
            s.print("value='" + val + "'>" + val + "<br>");
            s.println();
        }
        itemNo++; s.println("<br>");
    }

    public void printTextbox(String caption) {
        s.println(caption + "<br>");
        s.println("<textarea name='Q" + itemNo + "' cols='50' rows='3'></textarea><br>");
        itemNo++; s.println("<br>");
    }
}
```

```
<input type='radio' name='Q0'
    value='very much'>very much<br>
```

```
<input type='checkbox' name='Q1'
    value='Mathematics'>Mathematics<br>
```

```
<textarea name='Q2' cols='50' rows='3'>
</textarea><br>
```

Hauptprogramm

Aufgaben

- Liest Kommandozeilenparameter
- Erzeugt und initialisiert Scanner und Parser
- Startet den Parser

```
import java.io.*;
class MakeQueryForm {
   public static void main(String[] args) {
      String inFileName = args[0];
      String outFileName = args[1];
      Scanner scanner = new Scanner(inFileName);
      Parser parser = new Parser(scanner);
      try {
         parser.html = new HtmlGenerator(outFileName);
         parser.Parse();
         System.out.println(parser.errors.count + " errors detected");
      } catch (FileNotFoundException e) {
         System.out.println("-- cannot create file " + outFileName);
      }
   }
}
```

Übersetzung und Ausführung

ATG mit Coco/R übersetzen

```
java -jar Coco.jar QueryForm.ATG
```

⇨ Scanner.java, Parser.java

Alles übersetzen

```
javac Scanner.java Parser.java HtmlGenerator.java MakeQueryForm.java
```

Ausführen

```
java MakeQueryForm input.txt output.html
```

7. Compilergeneratoren

7.1 Überblick

7.2 Coco/R

7.3 Beispiele

- Fragebogen-Generator
- Lesen eines Binärbaums
- Erzeugung abstrakter Syntaxbäume

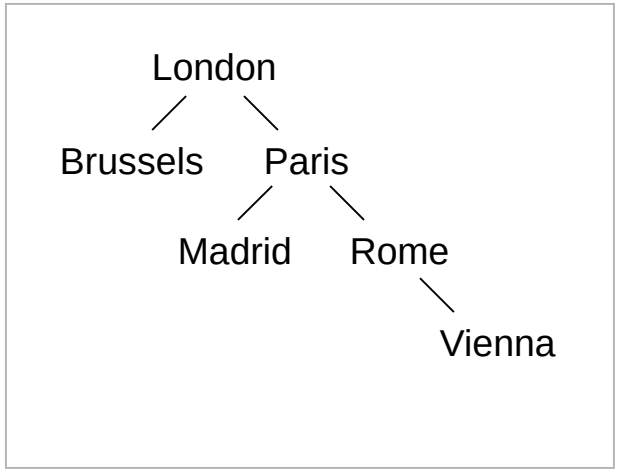
Lesen eines Binärbaums

Gegeben: Textdatei mit Binärbaum in Klammernotation

Textdatei

```
(London
   (Brussels)
   (Paris
      (Madrid)
      (Rome
         ()
         (Vienna)
      )
   )
)
```

Entspricht folgendem Baum



Gesucht:

- Einlesen der Textdatei
- Aufbau eines Binärbaums
- Ausgabe auf der Konsole

Knotenstruktur

```
class Node {
   String name;
   Node left;
   Node right;
}
```

Grammatik der Eingabe

Welche Muster müssen erkannt werden:

- ()
- (ident)
- (ident *Subtree Subtree*)

```
(London
   (Brussels)
   (Paris
      (Madrid)
      (Rome
         ()
         (Vienna)
      )
   )
)
```

Grammatik daher:

```
Subtree =
'('
   [ ident [ Subtree Subtree ] ]
')'.
```

Attribute

Subtree liefert einen *Node* als Wurzel des Unterbaums (kann *null* sein)

Semantische Routinen

print(node, indent);	gibt Baum mit Wurzel *node* im Eingabeformat auf der Konsole aus; Einrückungstiefe: *indent* Leerzeichen

Scannerbeschreibung


```
COMPILER TreeReader
CHARACTERS
   letter = 'A' .. 'Z' + 'a' .. 'z'.
TOKENS
   ident = letter {letter}.
IGNORE '\t' + '\r' + '\n'
...
END TreeReader.
```

Klammern müssen nicht als Tokens deklariert werden

Parserbeschreibung

```
COMPILER TreeReader

CHARACTERS
   letter = 'A' .. 'Z' + 'a' .. 'z'.

TOKENS
   ident = letter {letter}.

IGNORE '\t' + '\r' + '\n'

PRODUCTIONS

TreeReader                          (. Node n; .)
= Subtree<out n>                    (. print(n, 0); .) .

Subtree<out Node n>
= '('                               (. n = null; .)
   [ ident                          (. n = new Node(t.val); .)
      [   Subtree<out n.left>
          Subtree<out n.right>
      ]
   ]
   ')' .

END TreeReader.
```

Semantische Routine

```
COMPILER TreeReader
   class Node {
      String name;
      Node left, right;
      Node(String s) { name = s; }
   }
   static void print (Node n, int indent) {
      for (int i = 0; i < indent; i++) System.out.print(' ');
      System.out.print('(');
      if (n != null) {
         System.out.print(n.name);
         if (n.left != null || n.right != null) {
            System.out.println();
            print(n.left, indent + 2);
            print(n.right, indent + 2);
            for (int i = 0; i < indent; i++) System.out.print(' ');
         }
      }
      System.out.println(')');
   }
CHARACTERS
   ...
END TreeReader.
```

Ausgabe

```
(London
   (Brussels)
   (Paris
      (Madrid)
      (Rome
         ()
         (Vienna)
      )
   )
)
```

Hauptprogramm


```
class TreeReader {
   public static void main(String[] args) {
      Scanner scanner = new Scanner(args[0]);
      Parser parser = new Parser(scanner);
      parser.Parse();
      System.out.println(parser.errors.count + " errors detected");
   }
}
```

Übersetzen und ausführen

ATG mit Coco/R übersetzen:	**java -jar Coco.jar TreeReader.atg**
Java-Compiler laufen lassen:	**javac Scanner.java Parser.java TreeReader.java**
Ausführen:	**java TreeReader input.txt**

Zusammenfassung

Compilererzeugende Werkzeuge wie Coco/R sind immer dann nützlich ...

- wenn irgendeine Eingabe in eine Ausgabe transformiert werden soll
- wenn die Eingabe syntaktisch strukturiert ist

Typische Anwendungen

- Statische Programmanalyse
- Berechnung von Metriken aus Quellcode
- Instrumentierung von Quellcode
- Domänenspezifische Sprachen
- Analyse von Log-Dateien
- Verarbeitung von Datenströmen
- ...

7. Compilergeneratoren

7.1 Überblick

7.2 Coco/R

7.3 Beispiele

- Fragebogen-Generator
- Lesen eines Binärbaums
- Erzeugung abstrakter Syntaxbäume

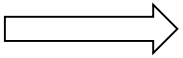
Erzeugung abstrakter Syntaxbäume

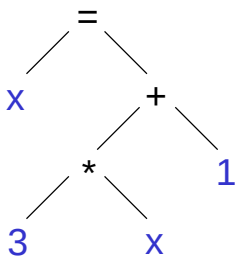
Abstrakter Syntaxbaum (AST)

- Blätter: Operanden
- innere Knoten: Operatoren

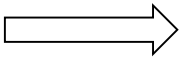
Beispiele

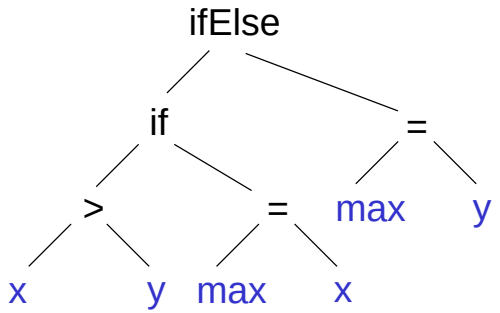
```
x = 3 * x + 1;
```






```
if (x > y) max = x; else max = y;
```



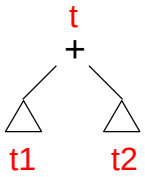


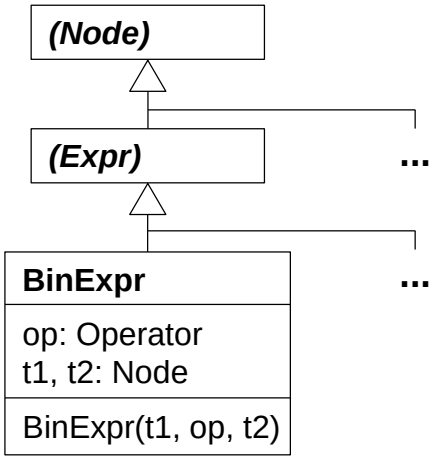
ASTs werden oft als interne Programmrepräsentation verwendet

Grundidee

Nonterminalsymbole liefern Teil-ASTs.
Diese werden zu einem neuen Teil-AST zusammengesetzt.

```
Expr ↑t = Term ↑t1 "+" Term ↑t2     (. t = new BinExpr(t1, PLUS, t2); .)
```





AST-Knoten sind Unterklassen von *Node*

Beispielsprache Taste

Sprache, für die wir ASTs bauen wollen

```
Taste     = "program" ident "{" { VarDecl | ProcDecl } "}".
VarDecl   = Type ident { "," ident } ";".
Type      = "int" | "bool".
ProcDecl  = "void" ident "(" ")" Block.

Block     = "{" { Stat | VarDecl } "}".
Stat      = ident ( "=" Expr ";"  |  "(" ")" ) ";"
          | "if" "(" Expr ")" Stat [ "else" Stat ]
          | "while" "(" Expr ")" Stat
          | "read" ident ";"
          | "write" Expr ";"
          | Block.

Expr      = SimExpr [ RelOp SimExpr ].
SimExpr   = Term { AddOp Term }.
Term      = Factor { MulOp Factor }.
Factor    = ident | number | "true" | "false" | "-" Factor.
RelOp     = "==" | "<" | ">".
AddOp     = "+" | "-".
MulOp     = "*" | "/".
```

Deklarationen

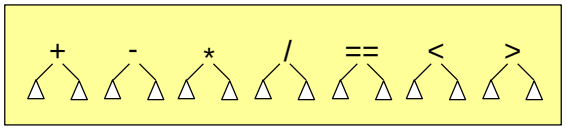
Anweisungen

Ausdrücke

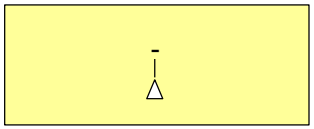
ASTs für Ausdrücke

Arten von Ausdrücken

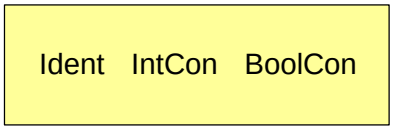
Binäre Ausdrücke



Unäre Ausdrücke



Blattknoten



Knotenklassen

```
abstract class Node {}

abstract class Expr extends Node {}

class BinExpr extends Expr {
    Operator op;
    Expr left, right;
    BinExpr (Expr e1, Operator op, Expr e2) {
        this.op = op; left = e1; right = e2;
    }
}

class UnaryExpr extends Expr {
    Operator op;
    Expr e;
    UnaryExpr (Operator op, Expr e) {
        this.op = op; this.e = e;
    }
}
```

```
class Ident extends Expr {
    Obj obj;
    Ident (Obj obj) { this.obj = obj; }
}

class IntCon extends Expr {
    int val;
    IntCon (int val) { this.val = val; }
}

class BoolCon extends Expr {
    boolean val;
    BoolCon (boolean val) { this.val = val; }
}
```

Obj ... siehe Deklarationen

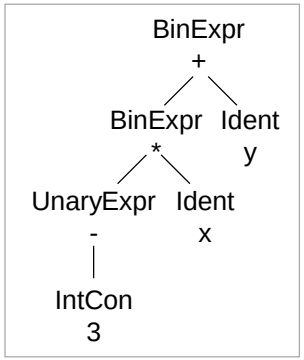
```
enum Operator {EQU, LSS, GTR, ADD, SUB, MUL, DIV}
```

Attributierte Grammatik

```
Expr<out Expr e>          (. Operator op; Expr e2; .)
= SimExpr<out e>
  [ RelOp<out op>
    SimExpr<out e2>       (. e = new BinExpr(e, op, e2); .)
  ].
SimExpr<out Expr e>       (. Operator op; Expr e2; .)
= Term<out e>
  { AddOp<out op>
    Term<out e2>          (. e = new BinExpr(e, op, e2); .)
  }.
Term<out Expr e>          (. Operator op; Expr e2; .)
= Factor<out e>
  { MulOp<out op>
    Factor<out e2>        (. e = new BinExpr(e, op, e2); .)
  }.
Factor<out Expr e>        (. String name; .)
= Ident<out name>         (. e = new Ident(curProc.find(name)); .)
| number                  (. e = new IntCon(Integer.parseInt(t.val)); .)
| "-" Factor<out e>       (. e = new UnaryExpr(Operator.SUB, e); .)
| "true"                  (. e = new BoolCon(true); .)
| "false"                 (. e = new BoolCon(false); .) .
```

```
Ident<out String name>
= ident   (. name = t.val; .) .
AddOp<out Operator op>
= "+"     (. op = Operator.ADD; .)
| "-"     (. op = Operator.SUB; .) .
MulOp<out Operator op>
= "*"     (. op = Operator.MUL; .)
| "/"     (. op = Operator.DIV; .) .
RelOp<out Operator op>
= "=="    (. op = Operator.EQU; .)
| "<"     (. op = Operator.LSS; .)
| ">"     (. op = Operator.GTR; .) .
```

- 3 * x + y



curProc.find(name) sucht name in Symbolliste

ASTs für Anweisungen

Var und *Proc*
... siehe Deklarationen

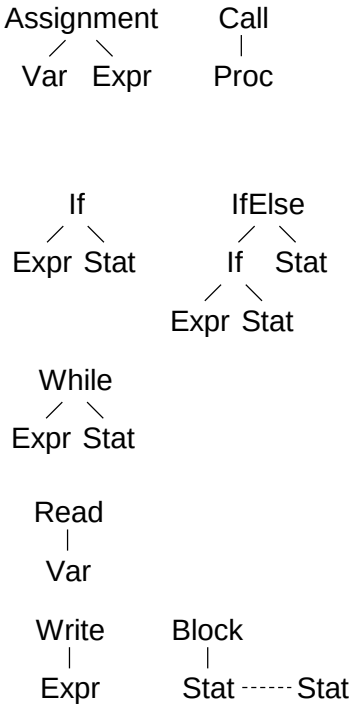
Knotenklassen

```
abstract class Stat extends Node {}

class Assignment extends Stat {
    Var left;
    Expr right;
    Assignment (Var v, Expr e) { left = v; right = e; }
}

class Call extends Stat {
    Proc proc;
    Call (Proc p) { proc = p; }
}

class If extends Stat {
    Expr cond;
    Stat stat;
    If (Expr e, Stat s) { cond = e; stat = s; }
}

class IfElse extends Stat {
    Stat ifPart;
    Stat elsePart;
    IfElse (Stat i, Stat e) { ifPart = i; elsePart = e; }
}
```

```
class While extends Stat {
    Expr cond;
    Stat stat;
    While (Expr e, Stat s) { cond = e; stat = s; }
}

class Read extends Stat {
    Var var;
    Read (Var v) { var = v; }
}

class Write extends Stat {
    Expr e;
    Write (Expr e) { this.e = e; }
}

class Block extends Stat {
    List<Stat> stats = new ArrayList<Stat>();
    void add(Stat s) { stats.add(s); }
}
```

Attributierte Grammatik

```
Block<out Block b>                (. Stat s; .)
= "{"                             (. b = new Block(); .)
  { Stat<out s>                   (. b.add(s); .)
  | VarDecl
  }
  "}" .

Stat<out Stat s>                  (. String name; Expr e; Stat s2; Block b; .)
= Ident<out name>                 (. Obj obj = curProc.find(name); .)
  ( "=" Expr<out e> ";"           (. s = new Assignment((Var)obj, e); .)
  | "(" ")" ";"                   (. s = new Call((Proc)obj); .)
  )
| "if" "(" Expr<out e> ")"
  Stat<out s>                     (. s = new If(e, s); .)
  [ "else" Stat<out s2>           (. s = new IfElse(s, s2); .)
  ]
| "while" "(" Expr<out e> ")"
  Stat<out s>                     (. s = new While(e, s); .)
| "read"
  Ident<out name> ";"             (. s = new Read((Var)curProc.find(name)); .)
| "write"
  Expr<out e> ";"                 (. s = new Write(e); .)
| Block<out b>                    (. s = b; .) .
```

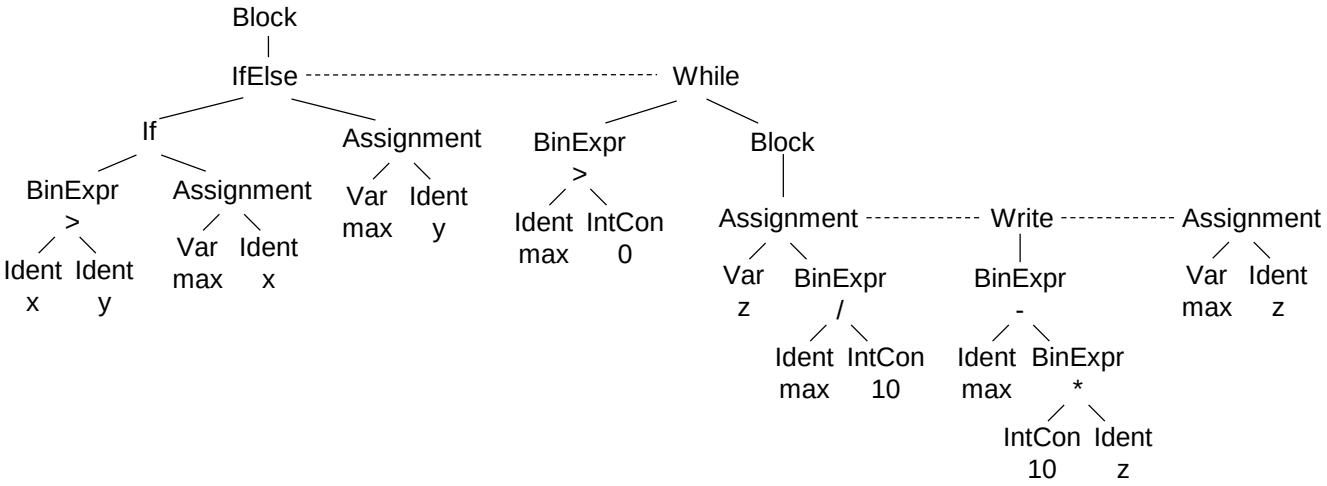


Beispiel: AST für Anweisungen

Taste-Programm

```
if (x > y) max = x; else max = y;
while (max > 0) {
   z = max / 10;
   write max - 10 * z;
   max = z;
}
```

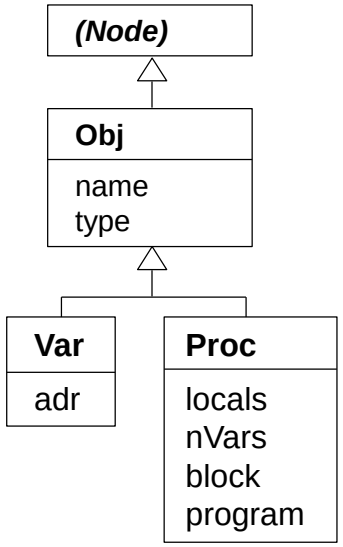
Abstrakter Syntaxbaum



ASTs für Deklarationen

Objektarten: *Var* und *Proc*

Symbolliste wird direkt im AST gespeichert (in *Proc*-Knoten)



Knotenklassen

```
class Obj extends Node {
    String name;
    Type type;
    Obj (String s, Type t) {
        name = s; type = t;
    }
}
class Var extends Obj {
    int adr;
    Var (String name, Type type) {
        super(name, type);
    }
}
```

```
class Proc extends Obj {
    List<Obj> locals = new ArrayList<>();
    int nVars = 0;
    Block block;     // statements
    Proc program; // program or null

    Proc (String name, Proc program) {
        super(name, Type.VOID);
        this.program = program;
    }

    void insert (Obj obj) { ... }

    Obj find (String name) { ... }
}
```

```
enum Type { VOID, INT, BOOL }
```

HauptProgramm ist ebenfalls ein *Proc*-Knoten

Symbollistenverwaltung


```
class Proc extends Obj {
    List<Obj> locals = new ArrayList<>();
    int nVars = 0;
    ...
    void insert (Obj obj) {
        for (Obj x: locals) {
            if (x.name.equals(obj.name)) SemErr(obj.name + " declared twice");
        }
        locals.add(obj);
        if (obj instanceof Var) ((Var)obj).adr = nVars++;
    }
    Obj find (String name) {
        for (Obj x: locals) { if (x.name.equals(name)) return x; }
        if (program != null) {
            for (Obj x: program.locals) { if (x.name.equals(name)) return x; }
        }
        SemErr(name + " undeclared");  // name not found
        return new Obj("_undef", Type.INT); // error object
    }
}
```

Attributierte Grammatik

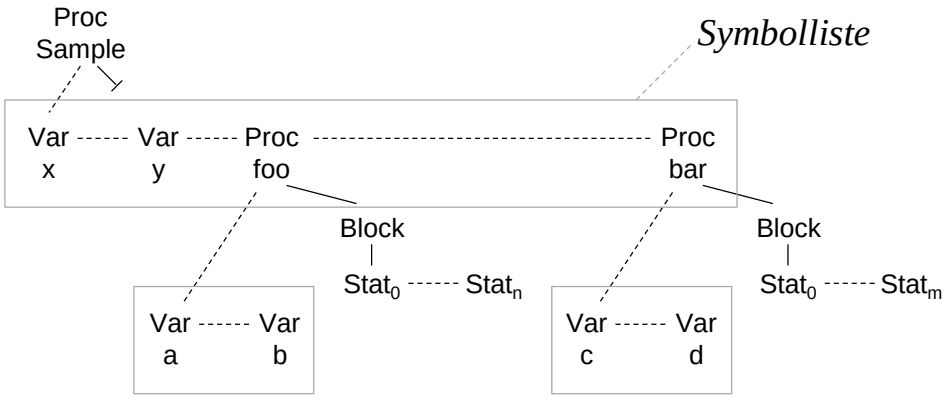

```
Taste                                (.  String name; .)
=  "program" Ident<out name>         (.  curProc = new Proc(name, null); .)
   "{"
   { VarDecl | ProcDecl }
   "}" .

VarDecl                              (.  String name; Type type; .)
=  Typ<out type>
   Ident<out name>                   (.  curProc.insert(new Var(name, type)); .)
   { "," Ident<out name>             (.  curProc.insert(new Var(name, type)); .)
   }
   ";" .

Typ<out Type type>
=  "int"                             (.  type = Type.INT; .)
|  "bool"                            (.  type = Type.BOOL; .) .

ProcDecl                             (.  String name; .)
=  "void" Ident<out name>            (.  Proc program = curProc;
                                         curProc = new Proc(name, program);
                                         program.insert(curProc); .)
   "(" ")"
   Block<out curProc.block>          (.  curProc = program; .) .
```

Beispiel: AST für Deklarationen

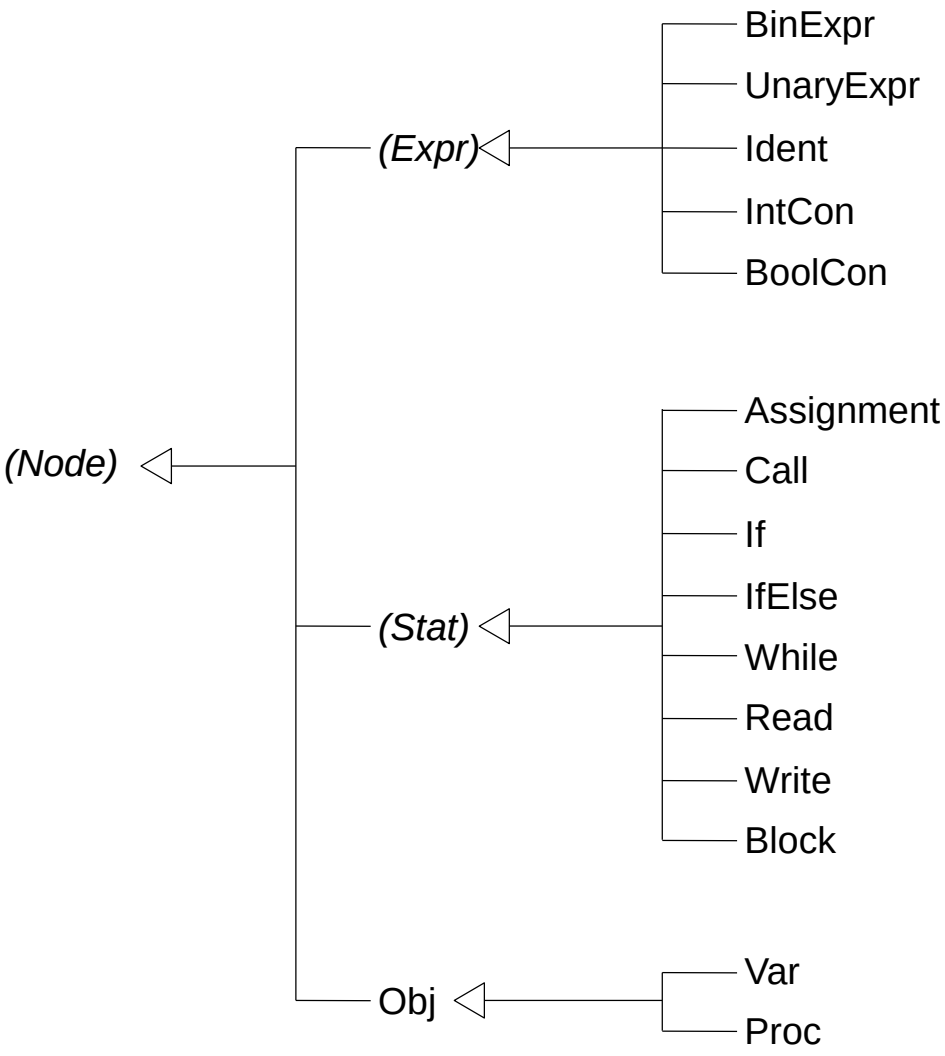
Taste-Programm

```
program Sample {
    int x;
    bool y;
    void foo() { int a, b; ... }
    void bar() { int c, d; ... }
}
```

Abstrakter Syntaxbaum



Zusammenfassung Knotenklassen



Compilerbeschreibung


```
COMPILER Taste
   Proc curProc;  // current program unit (procedure or main program)
CHARACTERS
   letter     = 'A' .. 'Z' + 'a' .. 'z'.
   digit      = '0' .. '9'.
TOKENS
   ident      = letter {letter | digit}.
   number     = digit {digit}.
COMMENTS FROM "//" to "\r\n"
IGNORE '\t' + '\r' + 'n'
PRODUCTIONS
   ...  // productions as described above
END Taste.
```

Hauptprogramm

```
class Taste {

   public static void main (String[] arg) {
      Scanner scanner = new Scanner(arg[0]);
      Parser parser = new Parser(scanner);
      parser.Parse();
      System.out.println(parser.errors.count + " errors detected");
   }
}
```

```
java -jar Coco.jar Taste.atg
javac Scanner.java Parser.java Taste.java ...

java Taste inputFile.tas
```