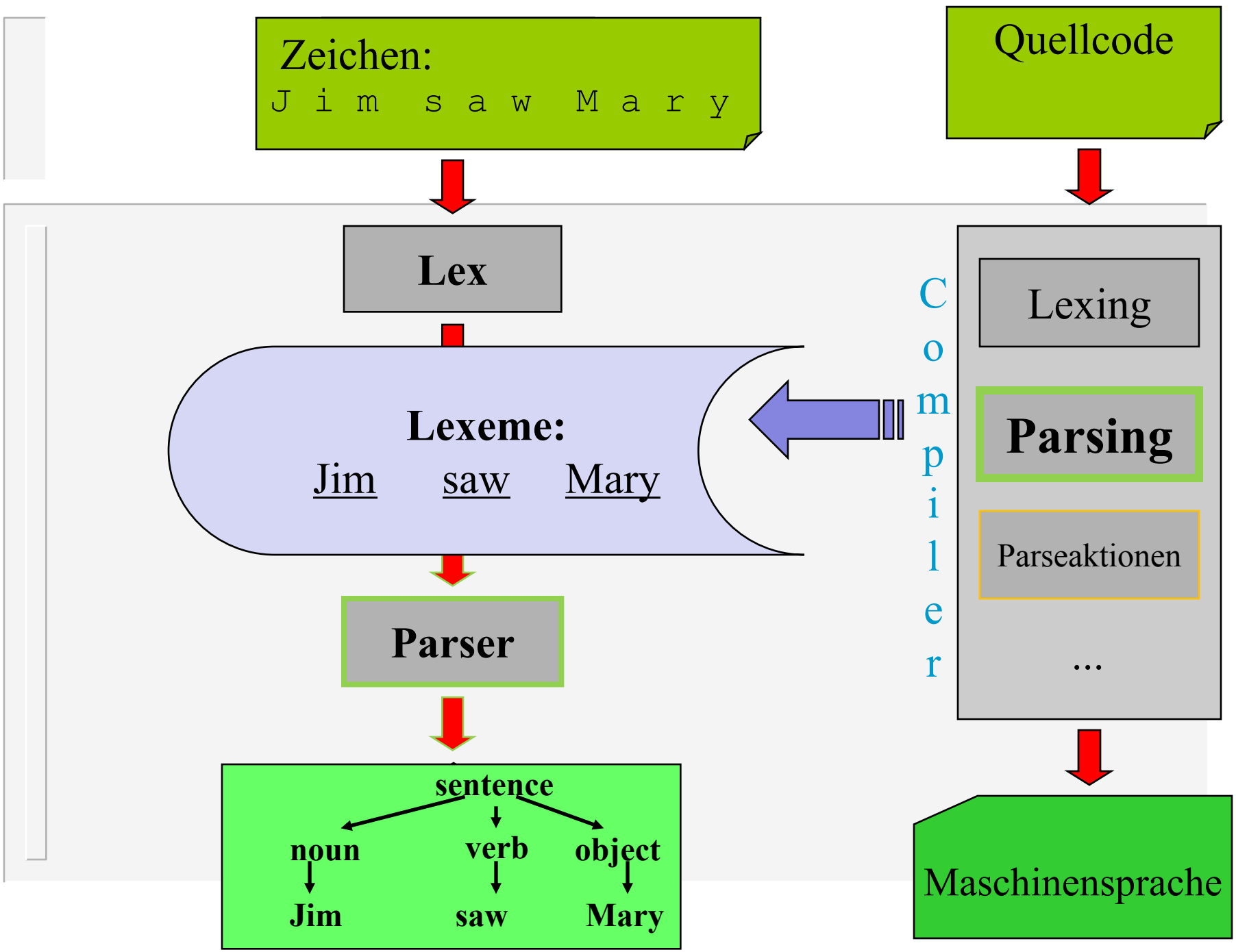
Kapitel 4

Semantische Aktionen und abstrakte Syntax

Michael Leuschel

John Witulski


Zeichen:
J i m s a w M a r y
Quellcode
Lex
Lexeme:
Jim saw Mary
Parser
sentence
noun
verb
object
Jim
saw
Mary
Compiler
Lexing
Parsing
Parseaktionen
...
Maschinensprache

Semantische Werte & Aktionen

- Idee: Füge den Knoten eines Parsebaums Werte hinzu
  - Terminale: Aufgabe
    - 52 ⇒ Num(52)
  - Nichtterminale: **Seman**
    - Verknüpft mit jeder Pro

Beispiel: Expr → Expr + Expr {
l = semantischer Wert des li
r = semantischer Wert des re
return l+r;

Für JavaCC tokens:

```
t = <ID>
{System.out.print(t.image);}
```

In JavaCC:

```
int Exp ():
{ int a,i; }
{  a=Term()
   ( "+" i=Term() {a=a+i;}
   | "-" i=Term() {a=a-i;}
   )*
 {return a; }
}
```

Semantische Regeln

Einfache Aktionen zur Berechnung semantischer Werte (auch Attribute genannt)

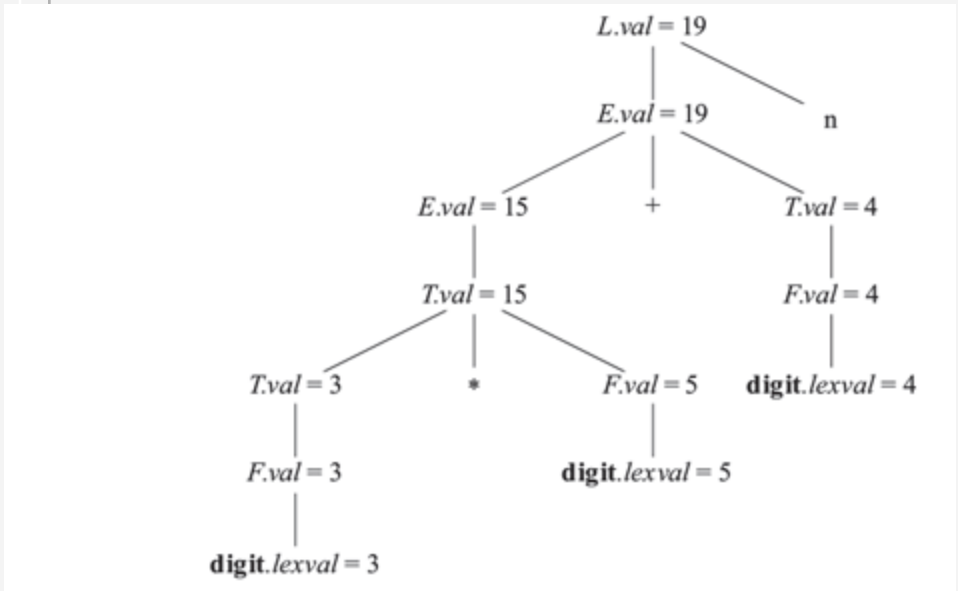


Abbildung 5.3: Kommentierter Parse-Baum für 3 * 5 + 4n

Produktion	Semantische Regeln
1) $L \rightarrow E$ n	$L.val = E.val$
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 * F$	$T.val = T_1.val$ x $F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow (E)$	$F.val = E.val$
7) $F \rightarrow$ digit	$F.val$ = digit.lexval

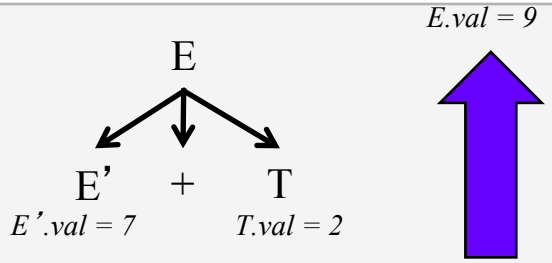
Abbildung 5.1: Syntaxgerichtete Definition eines einfachen Taschenrechners

Arten von Attributen

$E \rightarrow E' + T$

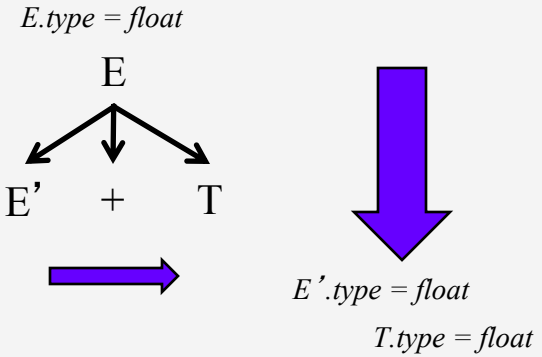
Synthetisiert:

- Attribut aus Attributen der Kinderknoten berechnet
- E.val = E'.val + T.val



Vererbt:

- Attribut aus Attributen von Eltern- oder Geschwisterknoten berechnet
- E'.type= E.type
- T.type = E.type



Abhängigkeitsgraph

Gibt an in welcher Reihenfolge Attribute berechnet werden.

Knoten: Attribute Kanten: „Hängt ab von“ Beziehung

z.B. E.b := E’.c neue Kante b → c

Notation:

Vererbte Attr. Symbol Synth. Attr.

- Wird über Parsebaum gezeichnet
- Synthetisierte Attribute rechts vom Knoten
- Vererbte Attribute links vom Knoten

Auswertungsreihenfolge von semantischen Aktionen:

- Topologische Sortierung auf dem Abhängigkeitsgraphen
- Der Graph muss ein DAG sein (keine Zyklen)

Arten von Attributen (2)

- synthetisiert (syn)
- vererbt (inh)

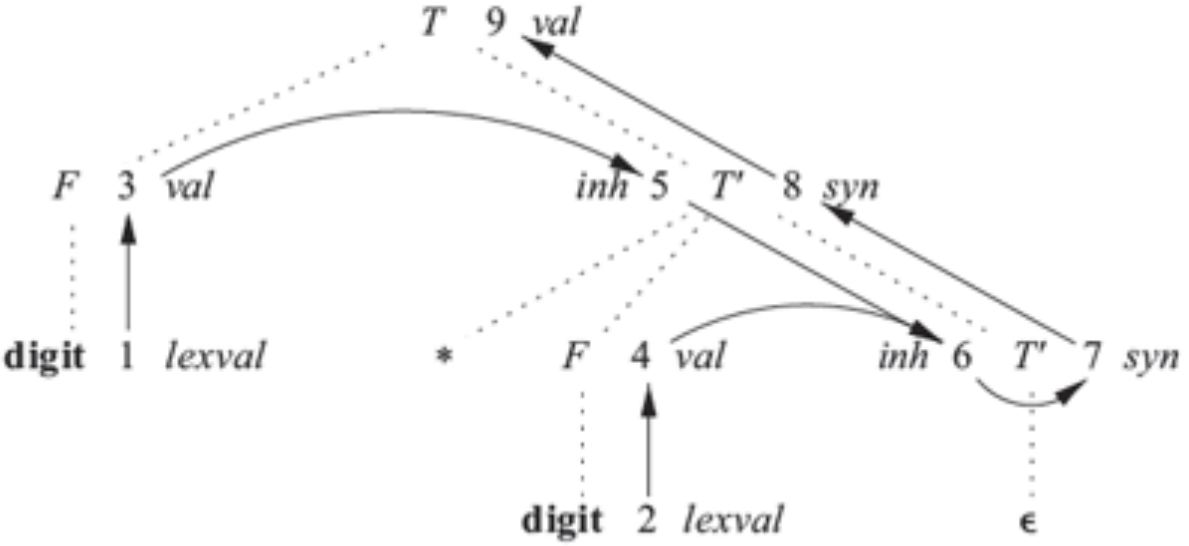


Abbildung 5.7: Abhängigkeitsgraph für den kommentierten Parse-Baum aus Abbildung 5.5

Arten von Attributgrammatiken

S-Attributgrammatiken:

Alle Attribute werden synthetisiert

E → AB {**E**.val := A.val + B.val}

L-Attributgrammatiken:

Nur erben von linken Geschwisterknoten

E → A**B** {B.val := A.val}

Beispiel: Interpreter

E → T+E {E.val = T.val+E.val}

E → T {E.val =T.val}

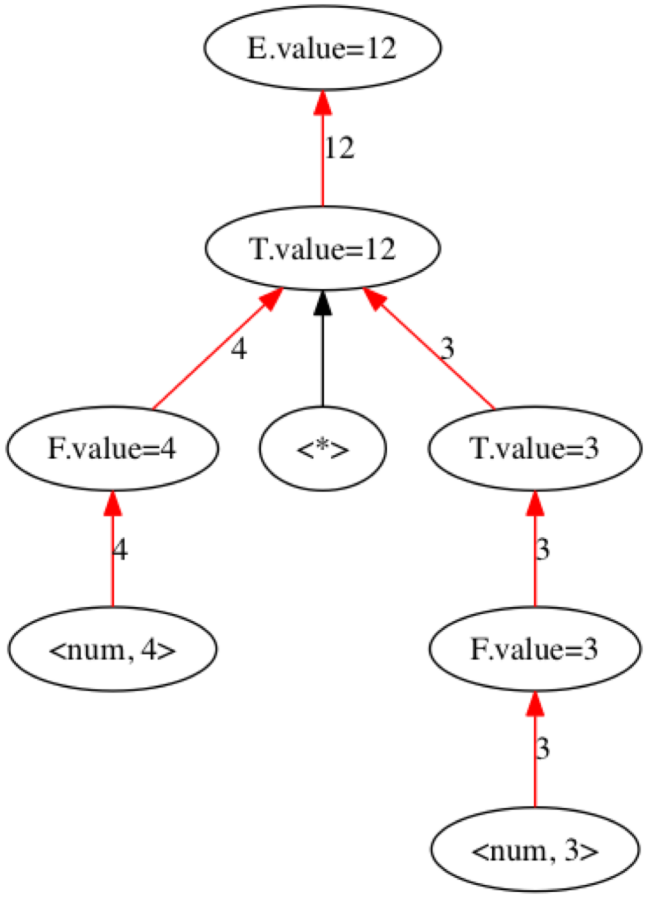
T → F*T {T.val = F.val*T.val}

T → F {T.val = F.val}

F → (E) {F.val = E.val}

F → Num {F.val = Num.lexval}

Eingabe: 4*3



Beispiel: AST-Generator

E → T+E {E.ast = new AddNode(T.ast, E.ast}

E → T {E.ast = T.ast}

T → F*T {T.ast = new MulNode(F.ast, T.ast}

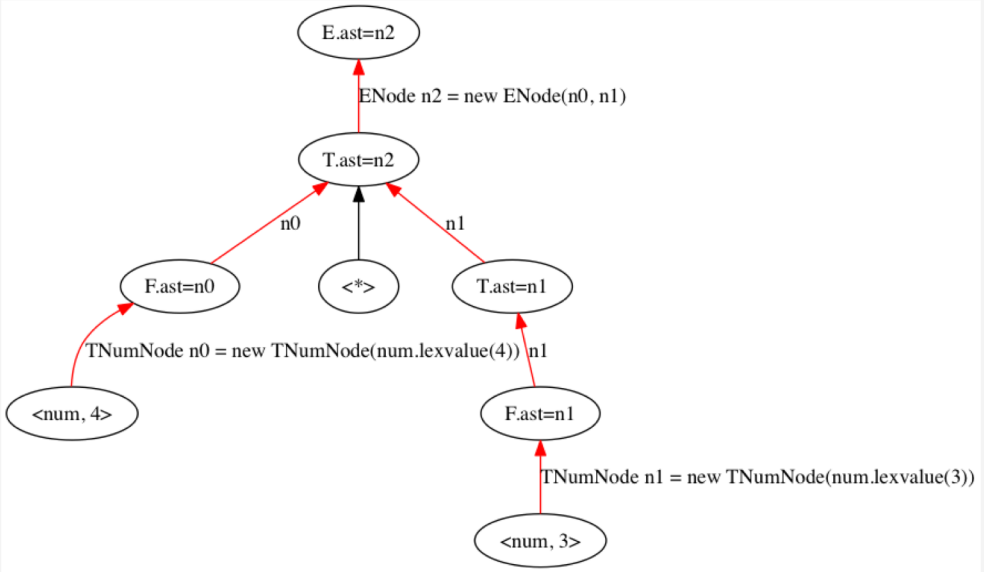
T → F {T.ast = F.ast}

F → (E) {F.ast = E.ast}

F → Num {F.ast = new

TNumNode(num.lexvalue) }

Eingabe: 4*3



Beispiel: Bytecode-Generator

E → T+E {E.code = T.code || E.code || iadd}

E → T {E.code = T.code}

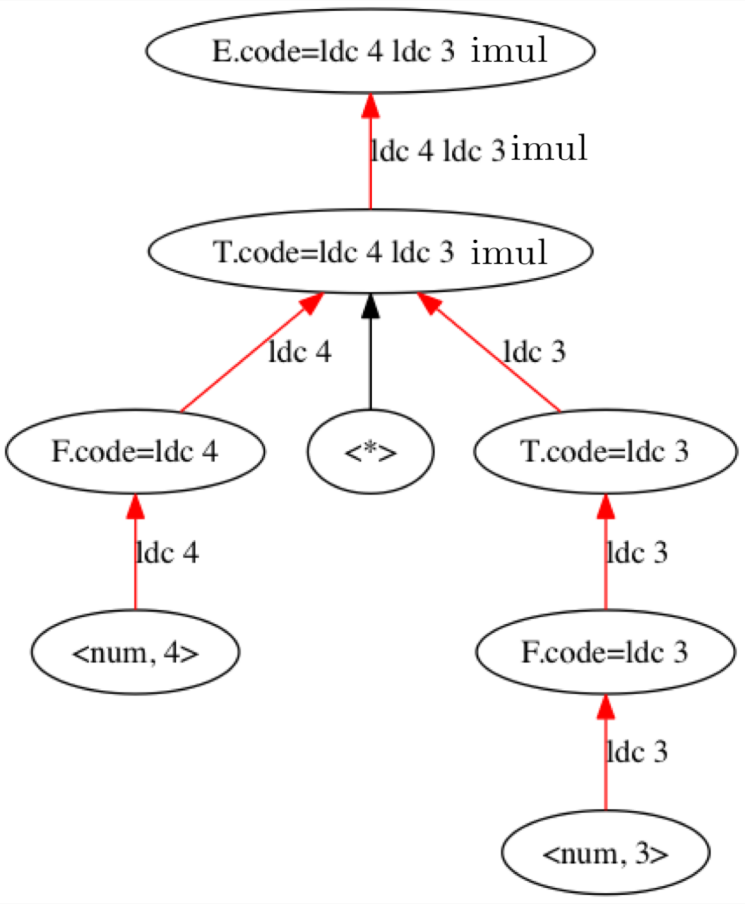
T → F*T {T.code = F.code || T.code || imul}

T → F {T.code = F.code}

F → (E) {F.code = E.code}

F → Num {F.code = ldc Num.lexvalue}

Eingabe: 4*3



Recursive Descent Parser: Hinzufügen von semantischen Aktionen

```
public static int Expr() throws java.io.IOException
    {int r; switch(tok) {
  case '(': /* Expr --> ( Expr ) */
 eat('('); r = Expr(); eat(')');
 return r;
  case '0': case '1': case '2': case '3': case '4':
  case '5': case '6': case '7': case '8': case '9':
 /* Expr --> Num */
 r = Num();
 return r;
  default: eat('(');  /* generiert Fehlermeldung */
    }
}
```

Semantische Aktionen in JavaCC

- Wird der rechten Seite (RHS) einer Produktion hinzugefügt

```
int Braces() :
{ int cnt=0; }
{
  <LBRACE> [ cnt=Braces() ] <RBRACE>
  { return ++cnt; }
}
```

- Wird ausgeführt wenn der Parser diesen Punkt erreicht
  - (Jedoch nicht beim Betrachten des Lookaheads)

Semantische Aktionen in CUP/Yacc

- Wird Produktionen hinzugefügt (nutze {: und :})

```
Expr ::= Num:n {: return n; :}
```

- Ausführung bei Reduktionen
- Verschiedene Werte möglich
  - Num,Expr: integer
  - Bexpr: boolean
- In SableCC:
  - Aufbau des AST (abstract syntax tree)

In CUP:

```
terminal Integer NUM;
non terminal Integer Expr;
non terminal Boolean Bexpr;
```

Ex → Num | (Ex) | Ex + Ex | Ex * Ex

Shift-Reduce Parsing

STACK	INPUT FILE	ACTION
	Num(2) + Num(2)	←shift
Num(2)	+ Num(2)	↓reduce 1
Ex(2)	+ Num(2)	←shift
Ex(2) +	Num(2)	←shift
Ex(2) + Num(2)		↓reduce 1
Ex(2) + Ex(2)		↓reduce 3
Ex(4)		SUCCESS

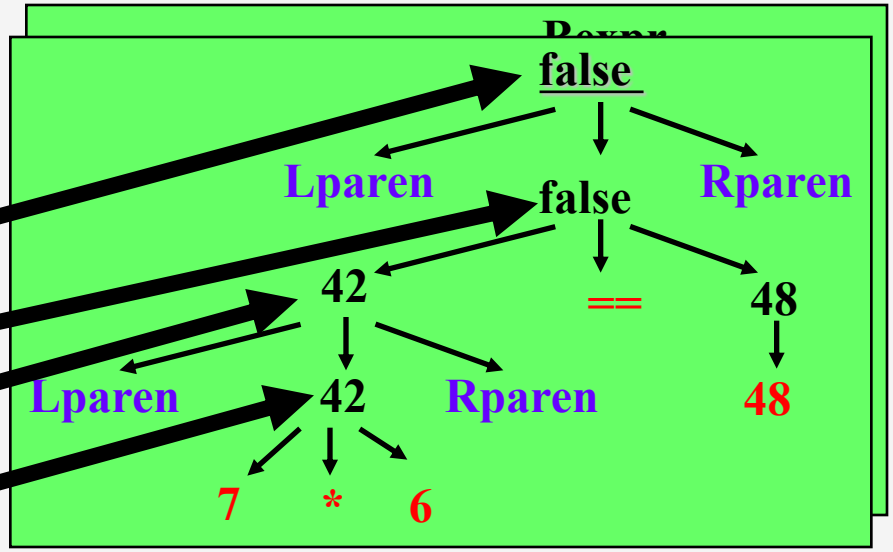
- ↓reduce 1: `Num:n {: RESULT=n; :}`
- ↓reduce 3:
```
Expr:l PLUS Expr:r
        {: RESULT= l + r; :}
```

Taschenrechner Beispiel

```
( ( 7 * 6)  ==  48) ;
```

Num ≡ (0|1|2|…|9)+
Mul ≡ “*”
Plus ≡ “+”
Eq ≡ “==“
Lparen ≡ “(“
Rparen ≡ “)”
Semi ≡ “;“

Lparen Lparen Num(7)
Mul Num(6) Rparen Eq
Num(48) Rparen Semi



Instr → Expr ; | Bexpr ;
Bexpr → Lparen Bexpr Rparen |
Expr Eq Expr
Expr → Num
Expr → Lparen Expr Rparen
Expr → Expr Plus Expr
Expr → Expr Times Expr

```
== false;
```

Minimal.lex

```
package Example2;
import java_cup.runtime.Symbol;
import java_cup.runtime.Scanner;
%%
%type Symbol
%function next_token
%implements Scanner
%cup
%%
";" { return new Symbol(sym.SEMI); }
"+" { return new Symbol(sym.PLUS); }
"*" { return new Symbol(sym.TIMES); }
"(" { return new Symbol(sym.LPAREN); }
")" { return new Symbol(sym.RPAREN); }
"==" { return new Symbol(sym.EQ); }
[0-9]+ { return new Symbol(sym.NUMBER, new Integer(yytext())); }
[ \t\r\n\f] { /* ignore white space. */ }
. { System.err.println("Illegal character: "+yytext()); }
```

Minimal.cup I

```
package Example2;
import java_cup.runtime.*;

parser code {:
public static void main(String args[]) throws Exception {
    new parser(new Yylex(System.in)).parse();
}
:}

terminal SEMI, PLUS, TIMES, LPAREN, RPAREN, EQ;
terminal Integer NUMBER;

non terminal instr_list, instr;
non terminal Integer expr;
non terminal Boolean bexpr;

precedence left EQ;
precedence left PLUS;
precedence left TIMES;
```

Minimal.cup II

```
instr_list ::= instr_list instr | instr;
instr ::= expr:e {: System.out.println(" = "+e+";"); :} SEMI |
  bexpr:e {: System.out.println(" == "+e+";"); :} SEMI ;
bexpr      ::= expr:l EQ expr:r
 {: RESULT=new Boolean(l.intValue()==r.intValue()); :}
    | bexpr:l EQ bexpr:r
 {: RESULT=new Boolean(l == r); :}
    | LPAREN bexpr:e RPAREN
 {: RESULT=e; :}
    ;
expr       ::= NUMBER:n
      {: RESULT=n; :}
             | expr:l PLUS expr:r
 {: RESULT=new Integer(l.intValue() + r.intValue()); :}
    | expr:l TIMES expr:r
 {: RESULT=new Integer(l.intValue() * r.intValue()); :}
    | LPAREN expr:e RPAREN
 {: RESULT=e; :}
    ;
```

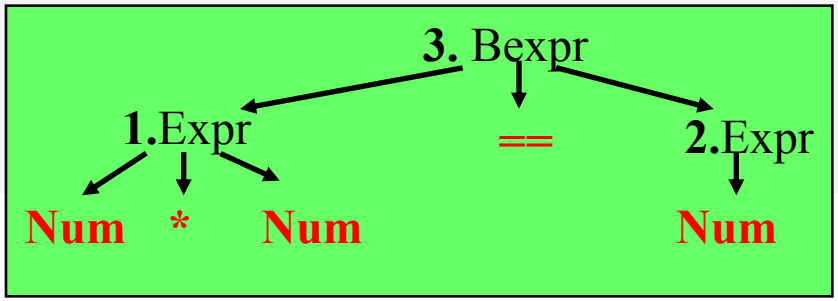
Ausführung von semantischen Aktionen

Reihenfolge in der **Aktionen** ausgeführt werden:

- Kein Seiteneffekt in **Aktionen** → don't care
  - `Result = l+r;`
- Mit Seiteneffekt → **do** care
  - `write(fileid,"INC R1");`
  - `write(fileid,"JNZ Lbl");`

Die Reihenfolge muss vorhersehbar sein

z.B. CUP: bottom-up, left-to-right traversal of parse tree



Nutzen von semantischen Aktionen

Interpreter:

- Ok (wie im Taschenrechner Beispiel)

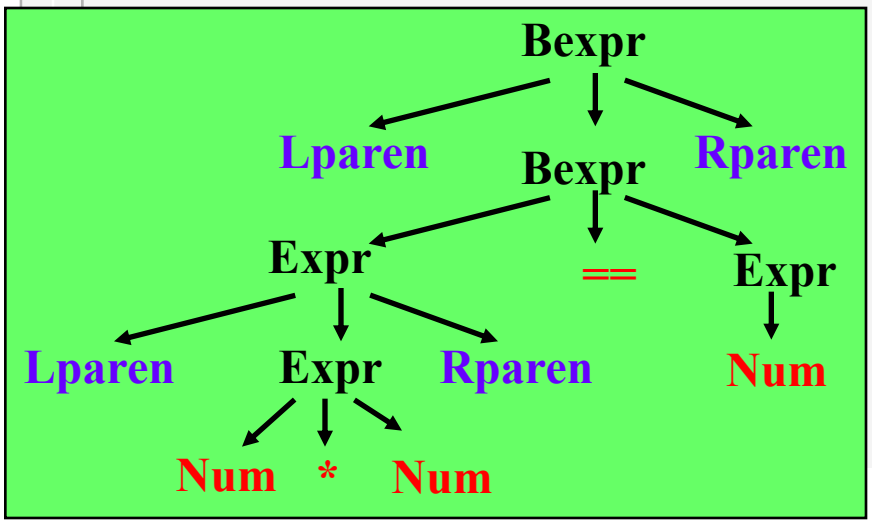
Compiler:

- Denkbar, aber:
  - Analysiere das Programm in der Parsereihenfolge!
  - Schwer zu schreiben, zu lesen und zu warten
    - z.B.: Variabel- oder Methodenaufruf vor Definition
    - Modularität (type checking, code generation,…),…
- In der Praxis:

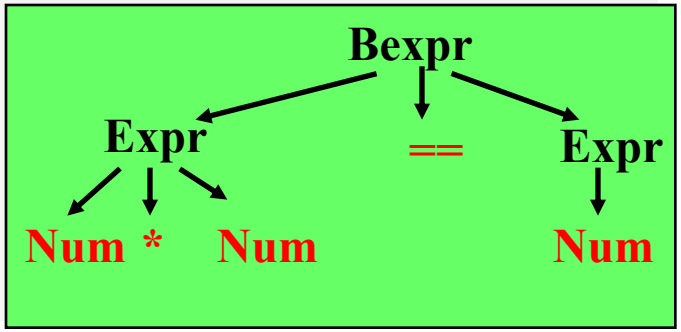
  Hiermit nur einen AST erzeugen

Abstrakter Syntaxbaum

- Konkreter Syntaxbaum
  - Enthält Trennzeichen (“(“, “)”, “;”, “begin”,…)
    - Ist beim parsen nützlich
    - Enthält nutzlose Informationen wenn der Baum aufgebaut wurde



⇒ abstrakte Grammatik & Baum



Grammatik für abstrakte Syntax

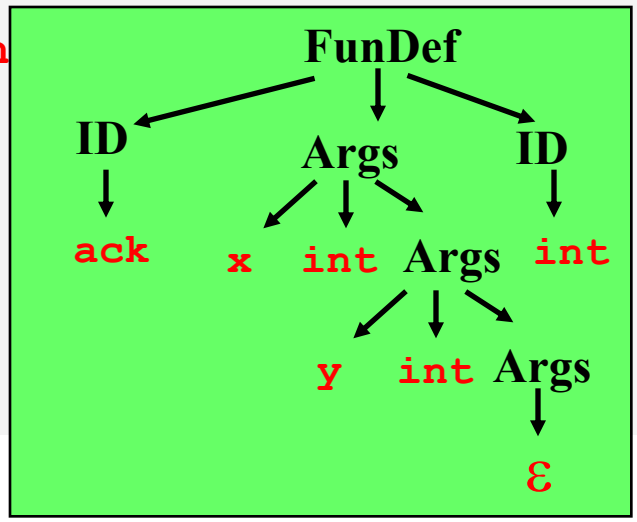
- Kann mehrdeutig sein
  - keine Trennzeichen
  - Für Parsing ungeeignet !

- FunDef = function ID (Args) : ID;
- Args = ε | ID: ID ; Args
- "`function ack(x:int; y:in`

Wird zu:

- FunDef = ID Args ID
- Args = ε | ID ID Args

→sauberes Interface



Noch ein Beispiel

- Mehrdeutige Grammatik

  Ex → Nat | ID | **(Ex)** | **Ex** + **Ex** | **Ex** * **Ex**

- Konkrete Parsingsyntax

  Ex → **Ex** + **Term** | **Term**
  Term → **Term** * **Factor** | **Factor**
  Factor → Nat | ID | **(Ex)**

  Ex → **Term Ex'**
  Ex' → ε | + **Ex**
  Term → **Factor T'**
  T' → ε | * **Term**
  Factor → Nat | ID | **(Ex)**

- Abstrakte Syntax

  Ex → Nat | ID | **Ex** + **Ex** | **Ex** * **Ex**

Abstrakter Syntaxbau

Eine Möglichkeit:

- 1 **abstrakte** Klasse pro Nicl
- 1 **konkrete** Klasse pro Regel
  - 1 Feld pro Nichtterminal auf rh
- (in abstrakter Grammatik)

Beispiel

- **Expr → Num**
- **Expr → Expr + Expr**

Für Anzeige von Fehlern:

```
public class Sum extends Expr {
  public Expr left,right;
  public FilePos start,end;
  public Sum(Expr l,r)
     {left = l; right = r;
      start = l.start;
      end = r.end;}
}
```

```
public abstract class Expr {}

public class Num extends Expr {
  public int val;
  public Num(int v) { val=v;}
}

public class Sum extends Expr {
  public Expr left,right;
  public Sum(Expr l,r)
     {left = l; right = r;}
}
```

Erzeugung von abstrakten Syntaxbäumen

- JavaCC
  - DIY durch semantische Aktionen
    - e1=Term() "+" e2=Term()
      {e1=new PlusExp(e1,e2);}
  - Oder nutze JJTree or JTB
- SableCC
  - LR Parsing
  - Keine semantischen Aktionen. Nur Aufbau des Parsebaums

Zusammenfassung

- **Semantische Werte und Aktionen/Regeln**
  - Wie werden:
    - Attribute Knoten hinzugefügt?
    - Regeln in Grammatiken modifiziert?
  - Nützlichkeit & Grenzen
- **Abstrakter Syntaxbaum**
  - Wieso und warum?
  - Wie wird dieser dargestellt?
  - Wie wird dieser konstruiert?

Was Sie nun können:

- **Front End eines Compilers**
- **Interpreter**

Neue Programmiersprachen erfinden:

Super-Python, Turbo-C, Java++,…

Domain specific language (DSL),...