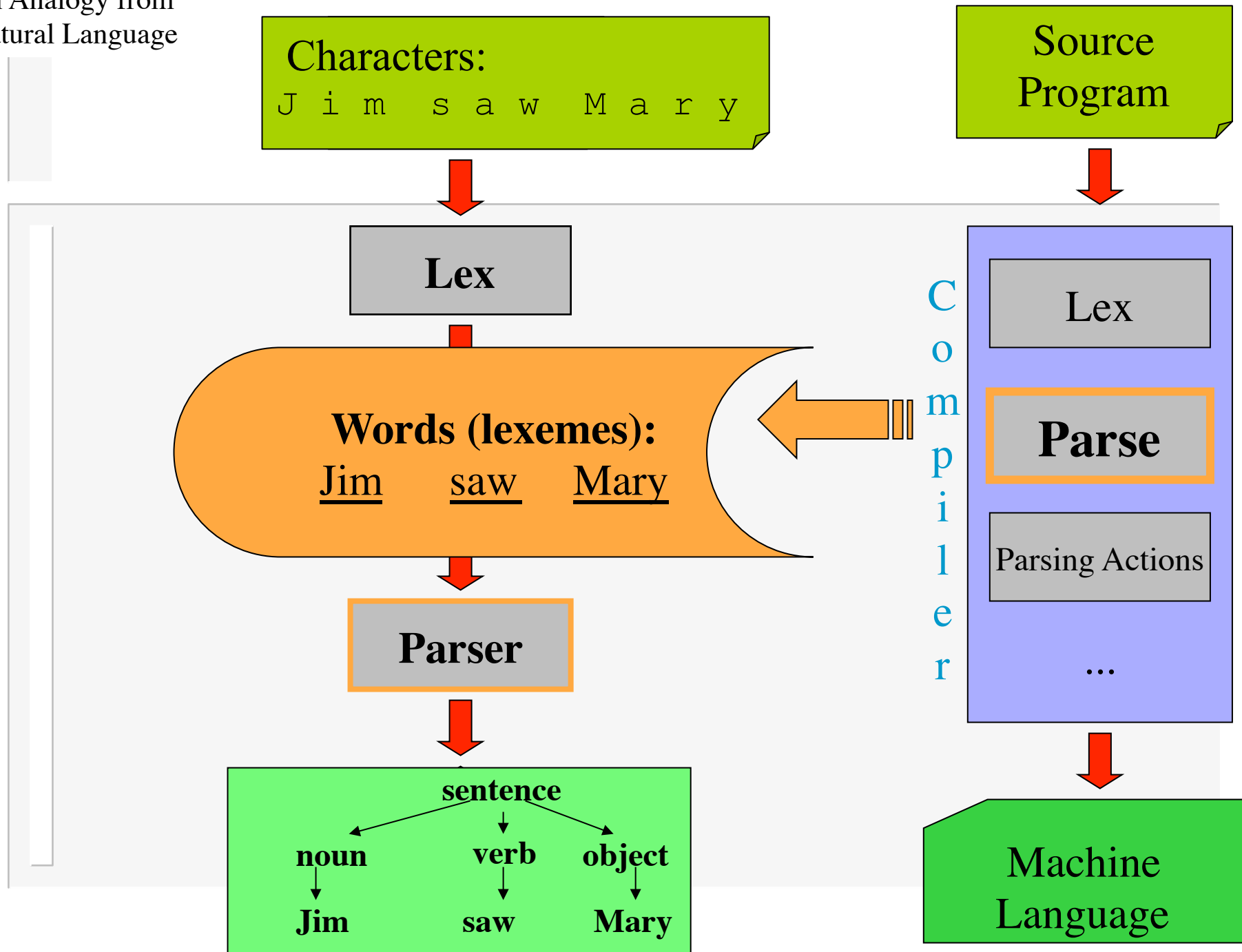


Kompilerverbau

**Semantische Aktionen &
Abstrakte Syntax**

Michael Leuschel

An Analogy from Natural Language



Semantic Values & Actions

- Idea: Attach values to symbols in the parse tree

- Terminals: done by

- 52 \Rightarrow Num(52)

- Non-Terminals: **Semantic Actions**

- associated with every production

- Example: $\text{Expr} \rightarrow \text{Expr} + \text{Expr}$

l = semantic value of left

r = semantic value of right

return l+r;

For 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 Ausrechnung semantischer Werte (auch Attribute genannt)

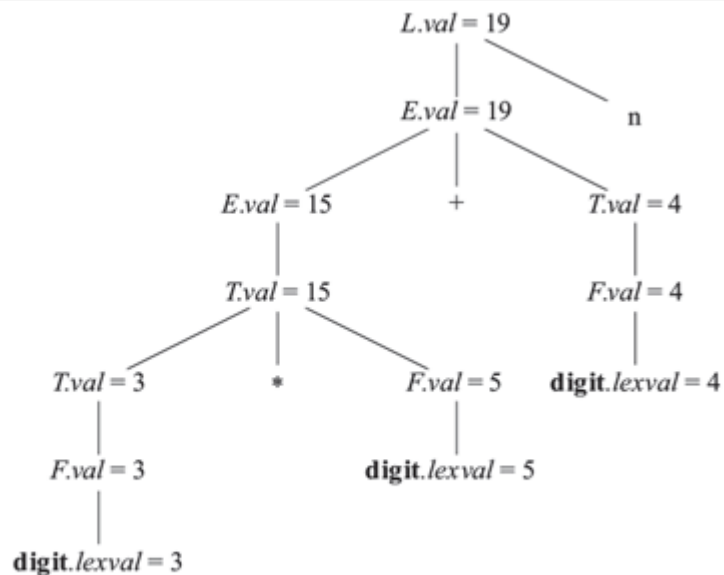


Abbildung 5.3: Kommentierter Parse-Baum für $3 * 5 + 4n$

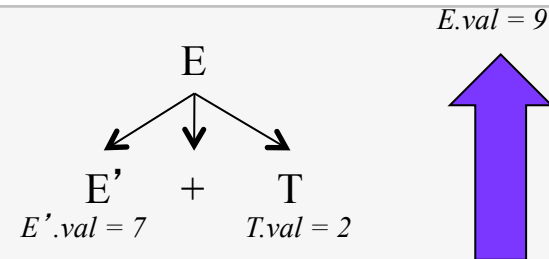
Produktion	Semantische Regeln
1) $L \rightarrow E n$	$L.val = E.val$
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow (E)$	$F.val = E.val$
7) $F \rightarrow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

Abbildung 5.1: Syntaxgerichtete Definition eines einfachen Taschenrechners

Arten von Attributen

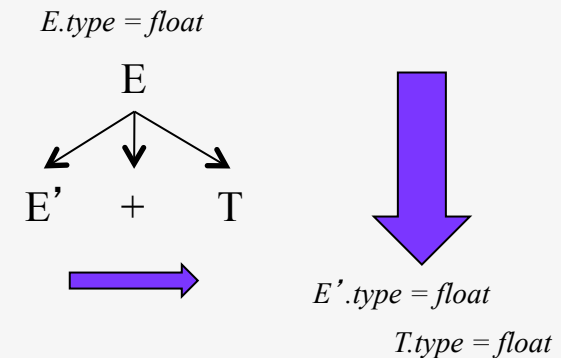
■ synthetisiert

- $E \rightarrow E' + T$
- $E.val = E'.val + T.val$



■ ererbt

- $E'.type = E.type$
- $T.type = E.type$



Arten von Attributen (2)

- synthetisiert
- ererbt

Details siehe Kapitel 5

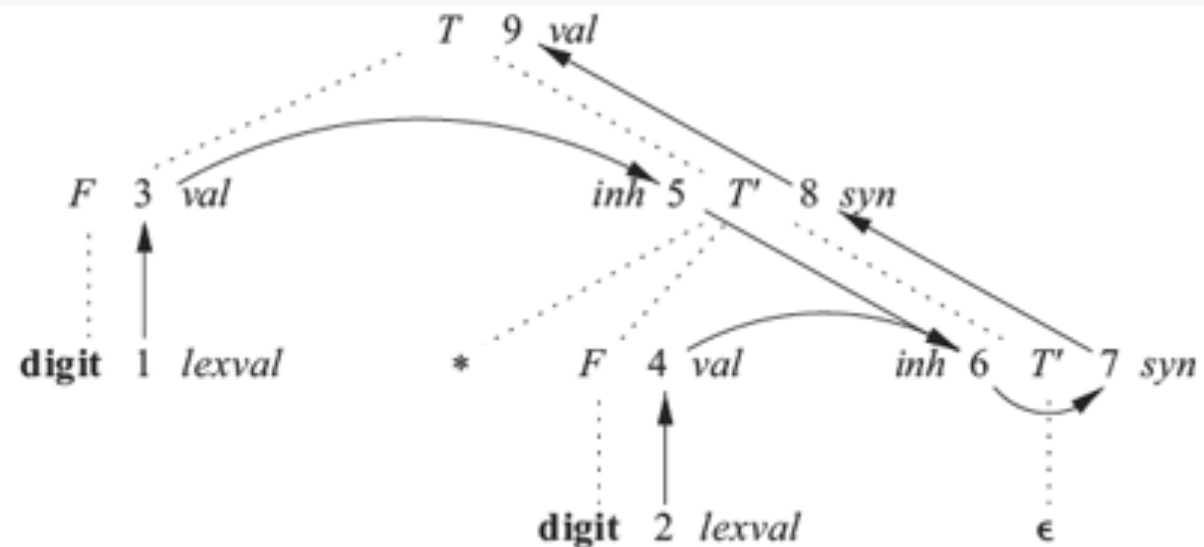


Abbildung 5.7: Abhängigkeitsgraph für den kommentierten Parse-Baum aus Abbildung 5.5

Adding Semantic Actions to Recursive Descent Parsers

```
1 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('('); /* generate error message */
   }
}
```

Semantic Actions in JavaCC

- Attached to points in RHS of productions

```
int Braces() :
```

```
{ int cnt=0; }
```

```
{
```

```
<LBRACE> [ cnt=Braces() ] <RBRACE>
```

```
  { return ++cnt; }
```

```
}
```

- Executed when parser reaches that point
 - (but not when examining lookahead)

Semantic Actions in CUP/Yacc

- Attached to productions (using {: and :})

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

- Executed when parser reduces
- Different types of values possible

- Num,Expr: integer
- Bexpr: boolean

- In SableCC:

- automatically builds (abstract syntax tree)

In CUP:

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

$$Ex \rightarrow \text{Num} \mid (Ex) \mid Ex + Ex \mid Ex * Ex$$

Shift-Reduce Parsing

<u>STACK</u>	<u>INPUT FILE</u>	<u>ACTION</u>
	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; : }

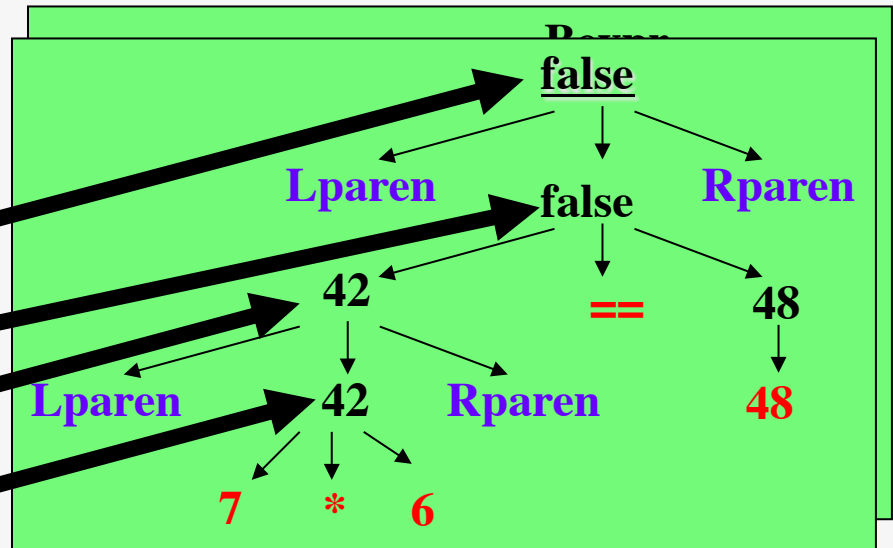
Calculator Example

((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;

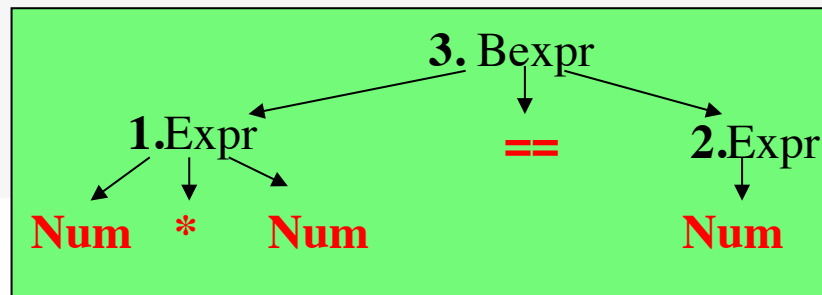
precedence left EQ;
precedence left PLUS;
precedence left TIMES;
```

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; : }
        ;
```

Execution Semantic Actions

- Order in which **actions** are performed
 - No side-effects in **actions** → don't care
 - `Result = l+r;`
 - With side-effects → **do** care
 - `write(fileid, "INC R1");`
 - `write(fileid, "JNZ Lbl");`
- Order of execution must be predictable
- CUP: bottom-up, left-to-right traversal of parse tree



Usefulness of Semantic Actions

- Interpreters:

- Ok (see our calculator example)

- Compilers:

- Possible, but:

- Analyze program in order it is parsed !

- Difficult to write, read, maintain

- Ex: Field, Method used before defined

- Modularity (type checking, code generation,...),...

- In practice:

- return **abstract syntax** tree for later phases

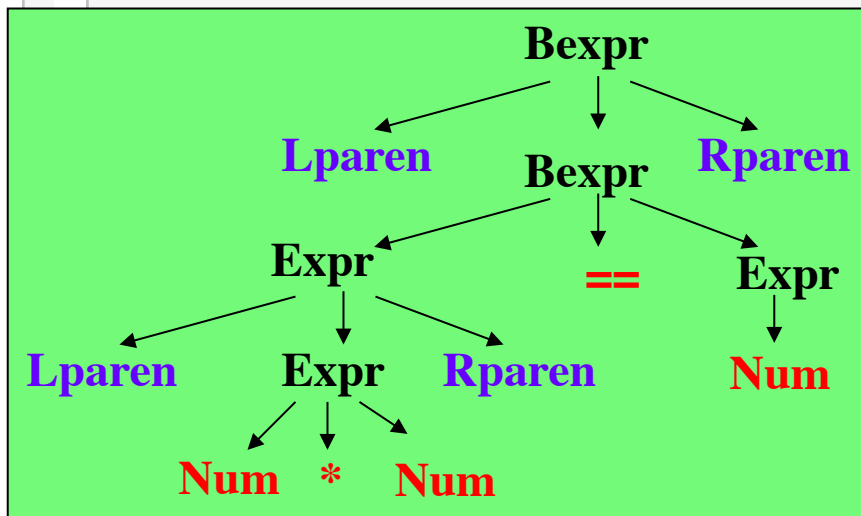
Abstract Syntax Tree

- Concrete syntax tree

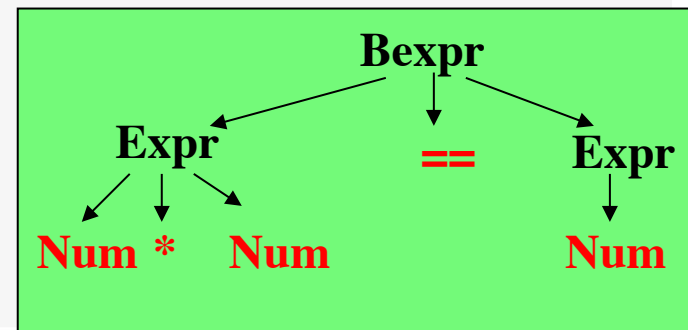
- Punctuation tokens (“(”, “)”, “;”, “begin”,...)

- Useful for parsing

- But no information once tree is built



⇒ abstract syntax grammar & tree



Grammar for Abstract Syntax

- Can be ambiguous, no punctuation symbols

- Not meant for parsing !

- FunDef = function ID (Args) : ID;

- Args = ϵ | ID: ID ; Args

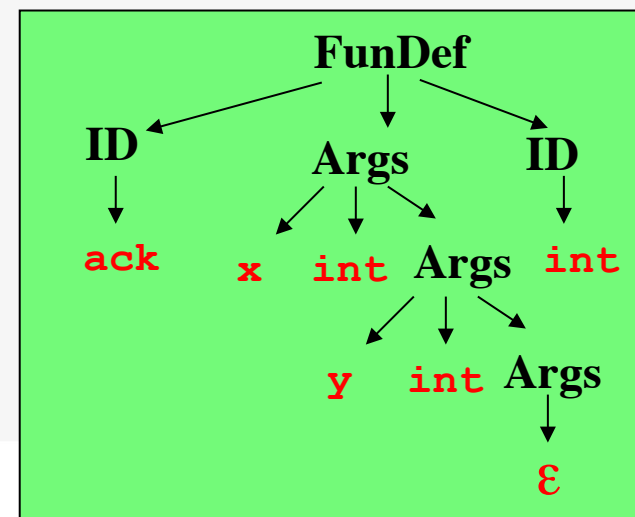
- "function ack(x:int; y:int;) : int;"

- Becomes

- FunDef = ID Args ID

- Args = ϵ | ID ID Args

→ Clean interface



Another Example

- Initial ambig. Grammar

$$Ex \rightarrow Nat \mid ID \mid (Ex) \mid Ex + Ex \mid Ex * Ex$$

- Concrete Parsing Syntax

$$\begin{aligned} Ex &\rightarrow Ex + Term \mid Term \\ Term &\rightarrow Term * Factor \mid Factor \\ Factor &\rightarrow Nat \mid ID \mid (Ex) \end{aligned}$$
$$Ex \rightarrow Term Ex'$$
$$Ex' \rightarrow \varepsilon \mid + Ex$$
$$Term \rightarrow Factor T'$$
$$T' \rightarrow \varepsilon \mid * Term$$
$$Factor \rightarrow Nat \mid ID \mid (Ex)$$

- Abstract Syntax

$$Ex \rightarrow Nat \mid ID \mid Ex + Ex \mid Ex * Ex$$

Abstract Syntax Tree

■ One way:

- 1 **abstract** class per non-terminal
- 1 **concrete** class per rule
 - 1 field per non-terminal on rhs
- (in abstract syntax grammar)

■ Example

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

Not really object-oriented
See Appel's book

For error reporting:

```
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;}  
}
```

Building Abstract Syntax Trees

■ JavaCC

- DIY via Semantic Actions
 - `e1=Term() "+" e2=Term()`
`{e1=new PlusExp(e1,e2);}`
- Or use JJTree or JTB

■ SableCC

- LR Parsing
- No semantic actions; but builds syntax tree automatically

Summary & What to know for the exam

■ **Semantic Values & Actions**

- How to associate
 - values to symbols and
 - actions to production rules
- Usefulness & limitations
- How to do them in JLex & CUP*

■ **Abstract Syntax Tree**

- Why and what
- How to represent them & construct them

* details have to be known for assignment but not exam

What you can do now:
- Front End of a compiler
- Interpreter

Brand new programming language:
HTML, XML, Java++,...
Domain specific language
for network configuration,...

