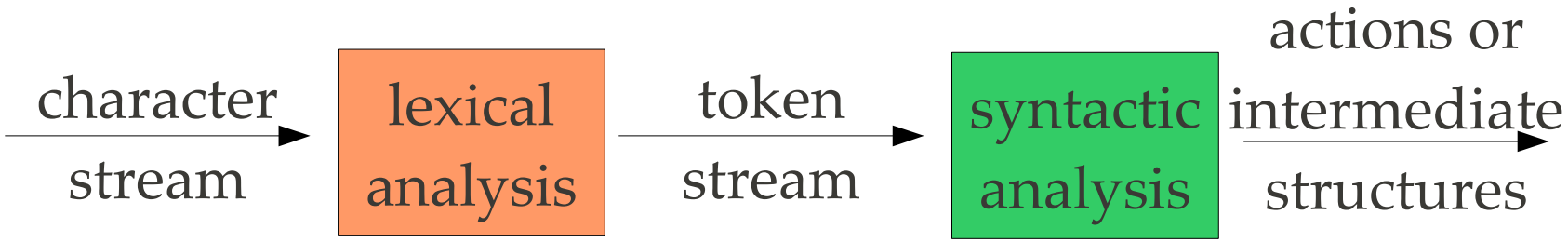
JavaCC

Ulrich.Schaefer@dfki.de

Parser generators

- goal: quickly write a parser for (non-XML) syntax
- idea: write down a CFG (in extended Backus-Naur form = EBNF) with associated actions, let a parser generator generate the parser instead of coding it manually (yacc + lex approach)
- easy to extend, modify, understand syntax(es)
- modularity: separate tokens from syntax from actions

LL(k), LR(k) syntax parsers

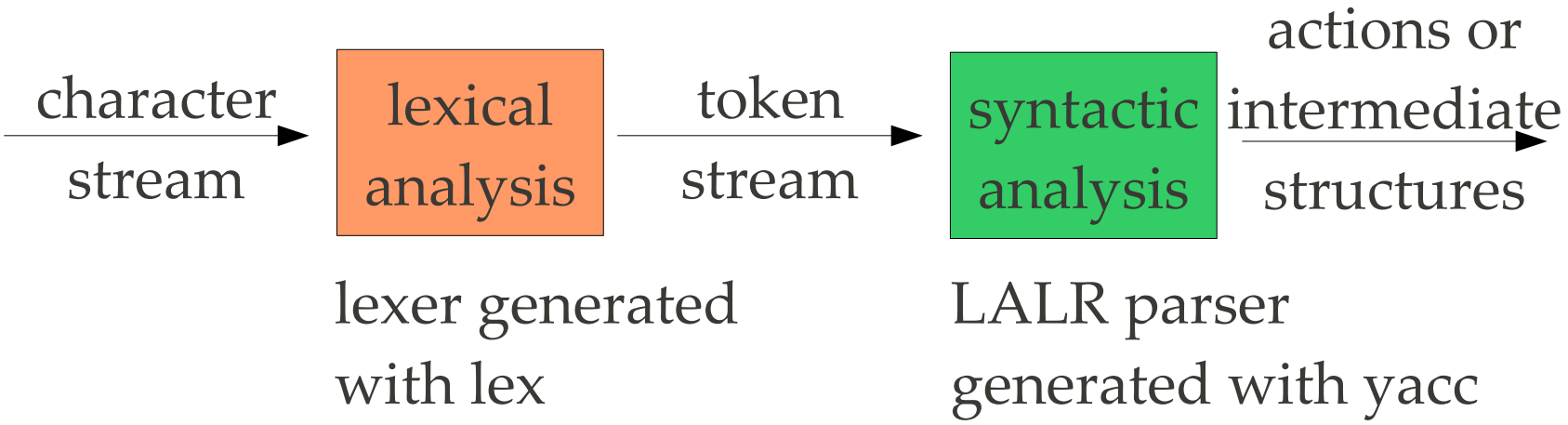


- LL(k): left-to-right, top-down
- LR(k): left-to-right, bottom-up
  - SLR: weaker than full LR, easy to implement
  - LALR: lookahead (efficient, compact, good trade-off)

Principles of parser generators

1. finite-state lexical analysis (tokenizer/lexer)
2. (LA)LR(k) or LL(k) parser
3. actions associated with grammar rules, defined in target programming language

UNIX lex and yacc (early 1970ies)



- e.g. used to implement C compilers
- finite state automata 1943 (McCulloch&Pitts)
- regular expressions/FSA 1956 (Kleene)
- context-free grammars 1956 (Chomsky)

Freely available Java parser generators

- there is no parser generator in Sun JDK
- **ANTLR**: http://www.antlr.org
  - ANother Tool for Language Recognition (formerly PCCTS)
  - LL(k), EBNF
  - excellent documentation
  - Java, C++, C# and Python code generation
- **SableCC**: http://sablecc.org
  - LALR(1), based on msc. thesis by Etienne Gagnon '98

JavaCC parser generator

- JavaCC home, download: http://javacc.dev.java.net
- authors: S. Sankar, S. Viswanadha
- open source (BSD license)
- grammars: LL(k), EBNF
- top-down parser generator (yacc: bottom-up)
- 100% Java, generates Java source code
- no additional jar file needed for runtime
- Unicode support
- exact position error reporting

JavaCC grammars

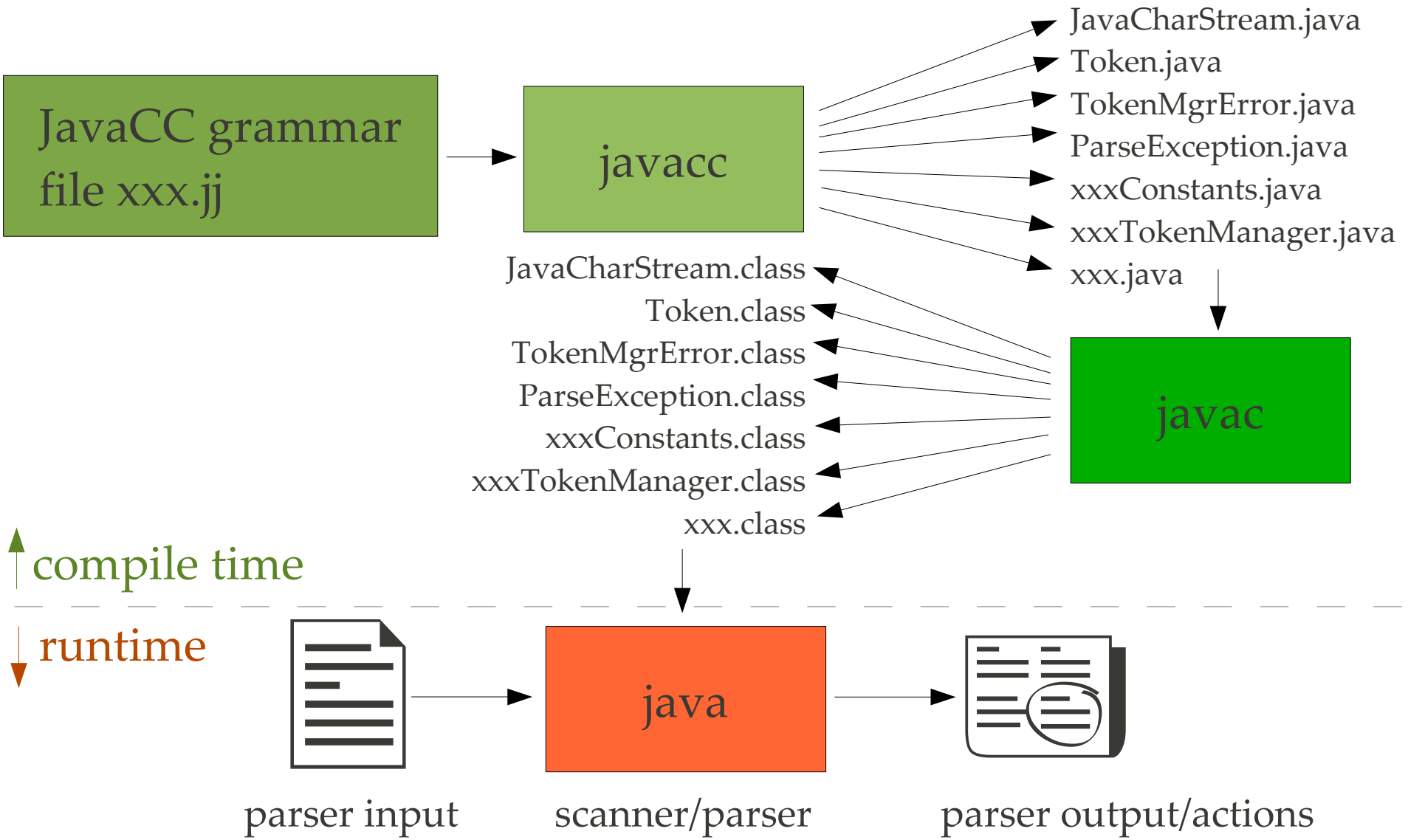
- single input file (lexer, grammar, actions)
  - Lexer: regular expressions
  - incorporates Java syntax in actions (JavaCC grammar syntax is defined in JavaCC! - Cf. syntax changes in Java 1.5 -> requires new JavaCC version 4)
- tools:
  - JJDoc generates grammar documentation (e.g., HTML)
  - JJTree inserts parse tree building code into grammar

Complex grammars developed in JavaCC

- Java
- JavaCC
- Javascript
- C
- Python/Jython
- XPath
- XQuery
- XML
- XML DTD
- HTML
- IDL
- RTF
- ASN.1
- Visual Basic, ...

- lexer ("tokenizer") definition:
  - regular expression syntax
  - Unicode
- parser grammar:
  - EBNF syntax
  - parameters
- actions:
  - Java code attached to EBNF

Generating, Compiling and Running a Parser



.jj grammar file syntax

- Java syntax for actions augmented with special syntax for lexer, EBNF, options etc.:
  1. options section for code generation settings
  2. class definition section, e.g. with auxiliary methods
  3. tokens section (lexer automaton definitions)
  4. EBNF rules with parameters and associated actions (similar to Java class method definitions)

1. Parser options section

- options influence the generated parser code
- example:

```
options {
   STATIC = false;                    // generate static methods
   JAVA_UNICODE_ESCAPE = true;        // allow \uXXXX
   UNICODE_INPUT = true;              // default is false
   IGNORE_CASE = true;                // default is false
   LOOKAHEAD = 2;                     // default is 1
   ...
}
```

2. Parser class body definition section

- the code will become part of the generated parser class

```
PARSER_BEGIN(Parsername)
  package javaKurs2.vorlesung6.beispielparser;
  import java.io.*;
  class Parsername { // Parsername = name of generated parser class
    static void main(String []args)
    throws ParseException, TokenMgrError {   // exceptions to catch
      Parsername parser = new Parsername(System.in);
      parser.Start();              // Start = name of start production
    }
  }  // insert here auxiliary methods, field variables etc.
PARSER_END(Parsername)
```

3. Definition of lexical analyzer


```
SKIP: { // characters to skip (e.g. whitespace)
        " " | "\t" | "\r"
      }
TOKEN: { // define lexical categories that can
         // be used in parser
    < LF : "\n" >
  | < MINUS : "-" >
  | < PLUS : "+" >
  | < NUM : (<DIGIT>)+ ( "." (<DIGIT>)+ )? >
  | < DIGIT : ["0"-"9"] >
}
```

regular expressions

range

Class Token (access in rule actions)

- access to
  - String Token.image
  - Token Token.next
  - int Token.beginLine
  - int Token.endLine
  - int Token.beginColumn
  - int Token.endColumn

4. Parser EBNF rules (1)

- EBNF = CFG augmented by * , +, ?, |, [] on RHS
- note: (...)? is equivalent to [...], but not in TOKEN!
- Example ("chain" calculator):

EBNF:

```
Input : ([Expr] <LF>)+ | <EOF>
Expr  : <NUM> ((<PLUS> <NUM>) | (<MINUS> <NUM>))*
```

Sample input:

```
45 + 57 - 89 + 4 - 62 + 40 ↵
result: -5
```

4. Parser EBNF rules (2)

- rule names (non-terminal symbols) correspond to generated method names
- non-terminals may have parameters and may return a value (like Java methods)
- actions can be associated with any RHS element
- variables can be declared locally (per rule) or globally (in parser class section/ see 1.)
- variables can be assigned to RHS elements

- Rule Syntax Schema

```
returntype Rulename (argtype1 arg1, ...) :
  { pre-actions (e.g., variable declarations);
  }
{ // Rule RHS is EBNF over
  [var=]Rulename(args) or [var=]<Token> { post-action; }
  { optional post-actions (e.g., return a value);
  }
}
```

JavaCC Grammar ("chain" calculator) 1


```
options { UNICODE_INPUT = true; }

PARSER_BEGIN(chaincalc)

public class chaincalc {

  public static void main(String args[])
  throws ParseException {

    chaincalc parser = new chaincalc(System.in);
    parser.Input();
  }
}

PARSER_END(chaincalc)
```

JavaCC Grammar ("chain" calculator) 2


```
// Lexer definition

SKIP : { " " | "\r" | "\t" }

TOKEN :
{  <LF: "\n">
|  <MINUS: "-">
|  <PLUS: "+">
| < NUM: (<DIGIT>)+ ( "." (<DIGIT>)+ )? >
| < DIGIT : ["0"-"9"] >
}

// <EOF> is predefined token for "end of file"
```

JavaCC Grammar ("chain" calculator) 3


```
void Input() :
{
  double result = 0.0; //important: initialize because optional on RHS
}
{
  ([result=Expr()] <LF> { System.out.println("result: "+result); } )+
  | <EOF>                { System.exit(-1); }
}

double Expr() :
{
  Token tnumber;
  Token tnum = null; // important: initialize because optional on RHS
  double result;
}
{
  tnumber = <NUM> { result = Double.parseDouble(tnumber.image); }
  ( (<PLUS> tnum=<NUM>) { result+=Double.parseDouble(tnum.image); } |
   (<MINUS> tnum=<NUM>) { result-=Double.parseDouble(tnum.image); })*
  { return result;
  }
}
```

Running JavaCC

- unpack javacc-5.0 binary archive
- javacc-5.0/bin/javacc is a shortcut for

```
java -classpath javacc-5.0/bin/lib/javacc.jar javacc $*
```

- generate parser: javacc grammar.jj [options...]
- compile parser: javac *.java
- run parser: java grammar
- Eclipse: plugin or define a new run configuration
- integration in Apache ant: later lecture

Literature

- Aho, Sethi, Ullman: COMPILERS Principles, Techniques and Tools (1986)
- JavaCC homepage: http://javacc.dev.java.net
- Eclipse plugin: http://eclipse-javacc.sourceforge.net/
- documented examples in distribution archive
- http://www.engr.mun.ca/~theo/JavaCC-Tutorial
- http://www.engr.mun.ca/~theo/JavaCC-FAQ/
- newsgroup comp.compilers.tools.javacc