Parsing

Prof. Dr. Ralf Lämmel
Universität Koblenz-Landau
Software Languages Team

An EBNF for the 101companies System

```
company :
  'company' STRING '{' department* '}' EOF;

department :
  'department' STRING '{'
    ('manager' employee)
    ('employee' employee)*
    department*
  '}';

employee :
  STRING '{'
    'address' STRING
    'salary' FLOAT
  '}';

STRING  :   '"' (~'"')* '"';
FLOAT   :   ('0'..'9')+ ('.' ('0'..'9')+)?;
```

Nontermin

Terminal

Grouping

Repetition

Context-free syntax

Another
EBNF for the 101companies System

```
COMPANY     : 'company';
DEPARTMENT  : 'department';
EMPLOYEE    : 'employee';
MANAGER     : 'manager';
ADDRESS     : 'address';
SALARY      : 'salary';
OPEN        : '{';
CLOSE       : '}';
STRING      : '"' (~'"')* '"';
FLOAT       : ('0'..'9')+ ('.' ('0'..'9')+)?;
WS          : (' '|'\r'? '\n'|'\t')+;
```

Lexical (= token level) syntax

Parsing is a form of language processing. What's a language processor?

- A program that performs language processing:
  - Acceptor
  - Parser
  - Analysis
  - Transformation
  - ...

Language processing *patterns*

1. The *Chopper* Pattern
2. The *Lexer* Pattern
3. The *Copy/Replace* Pattern
4. The *Acceptor* Pattern
5. The *Parser* Pattern

manual patterns

6. The *Lexer Generation* Pattern
7. The *Acceptor Generation* Pattern
8. The *Parser Generation* Pattern
9. The *Text-to-object* Pattern
10. The *Parser Generation*^{2} Pattern
11. (The *Text-to-tree* Pattern)
12. (The *Tree-walk* Pattern)
13. The *Object-to-text* Pattern

generative patterns

Approximates
the *Lexer Pattern*

The *Chopper* Pattern

The *Chopper* Pattern

- Intent:

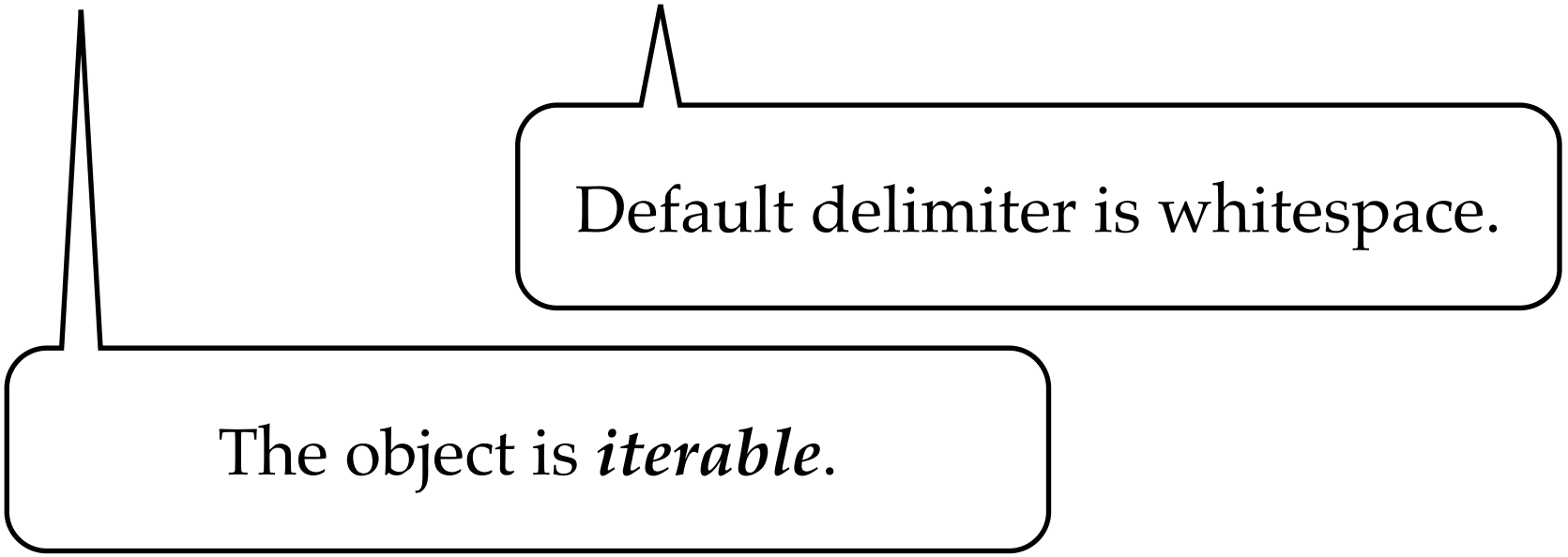
  Analyze text at the lexical level.

- Operational steps (run time):

  1. Chop input into “pieces”.
  2. Classify each piece.
  3. Process classified pieces in a stream.

Chopping input into pieces with java.util.Scanner

```
scanner = new Scanner(new File(...));
```



Tokens = classifiers of pieces of input

```
public enum Token {
   COMPANY,
   DEPARTMENT,
   MANAGER,
   EMPLOYEE,
   NAME,
   ADDRESS,
   SALARY,
   OPEN,
   CLOSE,
   STRING,
   FLOAT,
}
```

Classify chopped pieces into keywords, floats, etc.

```
public static Token classify(String s) {
  if (keywords.containsKey(s))
    return keywords.get(s);
  else if (s.matches("\"[^\"]*\""))
    return STRING;
  else if (s.matches("\\d+(\\.\\d*)?"))
    return FLOAT;
  else
    throw new RecognitionException(...);
}
```

Process token stream to compute salary total

```
public static double total(String s) throws ... {
   double total = 0;
   Recognizer recognizer = new Recognizer(s);
   Token current = null;
   Token previous = null;
   while (recognizer.hasNext()) {
      current = recognizer.next();
      if (current==FLOAT && previous==SALARY)
         total += Double.parseDouble(recognizer.getLexeme());
      previous = current;
   }
   return total;
}
```

The test for previous to be equal SALARY is not mandatory here. When could it be needed?

Demo

http://101companies.org/wiki/Contribution:**javaScanner**

Summary of implementation aspects

- Declare an enum type for tokens.
- Set up instance of `java.util.Scanner`.
- Iterate over pieces (strings) returned by scanner.
- Classify pieces as tokens.
  - Use regular expression matching.
- Implement operations by iteration over pieces.
  - For example:
    - Total: aggregates floats
    - Cut: copy tokens, modify floats

A problem with the *Chopper Pattern*

Input:
company “FooBar Inc.” { ...

Pieces:
‘company’, ‘“FooBar’, ‘Inc.”’, ‘{‘, ...

There is no general rule
for chopping the input into pieces.

Fixes
the *Chopper Pattern*

The *Lexer* Pattern

The *Lexer* Pattern

- Intent:

  Analyze text at the lexical level.

- Operational steps (run time):

  1. Recognize token/lexeme pairs in input.
  2. Process token/lexeme pairs in a stream.

Terminology

- *Token*: classification of lexical unit.
- *Lexeme*: the string that makes up the unit.

Lookahead-based decisions

```
  if (Character.isDigit(lookahead)) {
  // Recognize float
     ...
     token = FLOAT;
     return;
  }
  if (lookahead=='"') {
     // Recognize string
     ...
     token = STRING;
     return;
  }
...
```

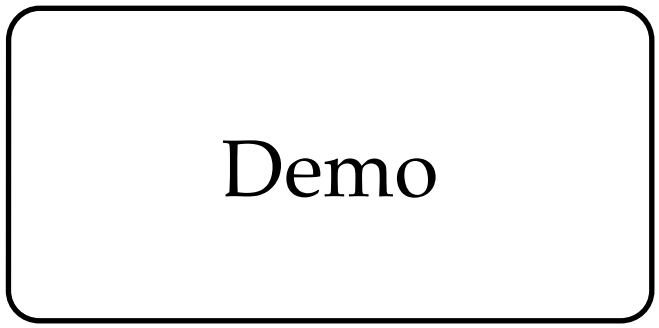
Inspect lookahead and decide on what token to recognize.

Recognize floats

```
if (Character.isDigit(lookahead)) {
  do {
    read();
  } while (Character.isDigit(lookahead));
  if (lookahead=='.') {
    read();
    while (Character.isDigit(lookahead))
      read();
  }
  token = FLOAT;
  return;
}
```

The code essentially implements this regexp:

```
"\\d+(\\.\\d*)?"
```



http://101companies.org/wiki/Contribution:**javaLexer**

Summary of implementation aspects

- Declare an enum type for tokens.
- Read characters one by one.
- Use lookahead for decision making.
- Consume all characters for lexeme.
- Build token/lexeme pairs.
- Implement operations by iteration over pairs.

Other approaches use automata theory (DFAs).

A problem with the *Lexer Pattern*
(for the concrete approach discussed)

```
if (Character.isDigit(lookahead)) {
    do {
        read();
    } while (Character.isDigit(lookahead));
    if (lookahead=='.') {
        read();
        while (Character.isDigit(lookahead))
            read();
    }
    token = FLOAT;
    return;
}
```

How do we get (back) the conciseness of regular expressions?

Builds on top of the *Lexer Pattern*

The *Copy/Replace* Pattern

The *Copy/Replace Pattern*

- Intent:

  Transform text at the lexical level.

- Operational steps (run time):

  1. Recognize token/lexeme pairs in input.
  2. Process token/lexeme pairs in a stream.
     1. Copy some lexemes.
     2. Replace others.

Precise copy for comparison

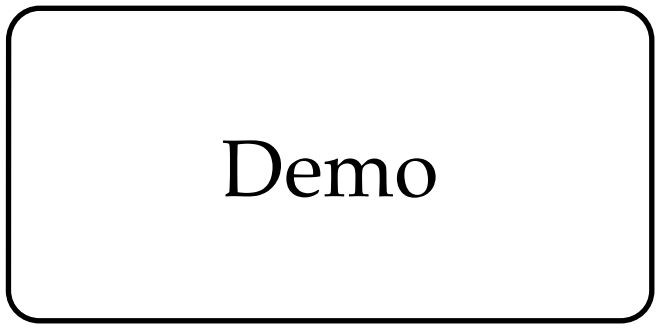
```
public Copy(String in, String out) throws ... {
   Recognizer recognizer = new Recognizer(in);
   Writer writer = new OutputStreamWriter(
           new FileOutputStream(out));
   String lexeme = null;
   Token current = null;
   while (recognizer.hasNext()) {
      current = recognizer.next();
      lexeme = recognizer.getLexeme();
      writer.write(lexeme);
   }
   writer.close();
}
```

Copy/replace for cutting salaries in half

```
...
    lexeme = recognizer.getLexeme();

    // Cut salary in half
    if (current == FLOAT && previous == SALARY)
       lexeme = Double.toString(
              (Double.parseDouble(
         recognizer.getLexeme()) / 2.0d));

    writer.write(lexeme);
...
```



http://101companies.org/wiki/Contribution:**javaLexer**

Summary of implementation aspects

- Build on top of the *Lexer Pattern*.
- Processor writes to an output stream.
- Processor may maintain history such as “previous”.

Builds on top of
the *Lexer Pattern*

The *Acceptor* Pattern

EBNF for lOlcompanies System

```
company :
  'company' STRING '{' department* '}' EOF;

department :
  'department' STRING '{'
    ('manager' employee)
    ('employee' employee)*
    department*
  '}';

employee :
  STRING '{'
    'address' STRING
    'salary' FLOAT
  '}';
```

Wanted: an acceptor

The *Acceptor* Pattern

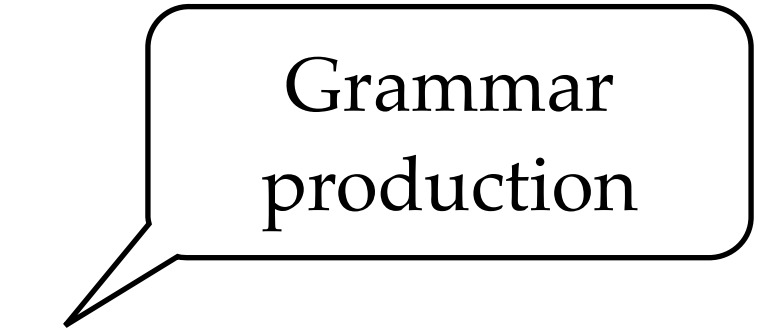
We assume a **recursive descent parser** as acceptor.

- Intent:

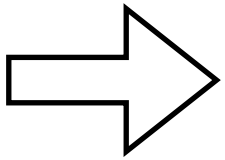
  Verify syntactical correctness of input.

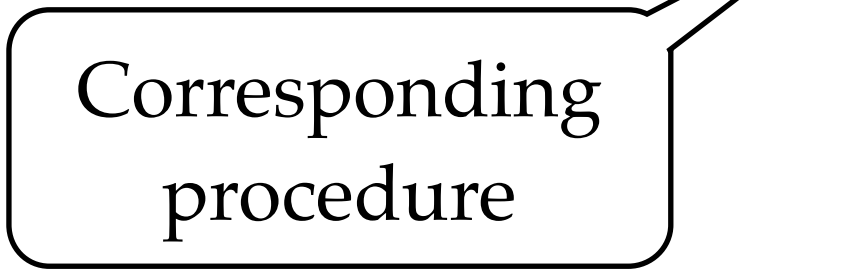
- Operational steps (run time):
  - Recognize lexemes/tokens based on the *Lexer Pattern*.
  - Match terminals.
  - Invoke procedures for nonterminals.
  - Commit to alternatives based on lookahead.
  - Verify elements of sequences one after another.
  - Communicate acceptance failure as exception.

Recursive descent parsing




```
department :
  'department' STRING '{'
    ('manager' employee)
    ('employee' employee)*
    dept*
  '}';
```





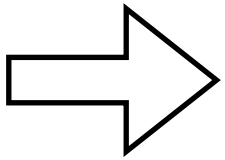

```
void department() {
   match(DEPARTMENT);
   match(STRING);
   match(OPEN);
   match(MANAGER);
   employee();
   while (test(EMPLOYEE)) {
      match(EMPLOYEE);
      employee();
   }
   while (test(DEPARTMENT))
      department();
   match(CLOSE);
}
```

Recursive descent parsing

A revised production

```
department :
  'department' STRING '{'
    ('manager' employee)
    ( ('employee' employee)
    | dept
    )*
  '}';
```

Use of alternatives



```
void department() {
   match(DEPARTMENT);
   match(STRING);
   match(OPEN);
   match(MANAGER);
   employee();
   while (test(EMPLOYEE)
       || test(DEPARTMENT)) {
     if (test(EMPLOYEE)) {
        match(EMPLOYEE);
        employee();
     } else
        department();
   }
   match(CLOSE);
}
```

Demo

http://101companies.org/wiki/Contribution:**javaParser**

See class Acceptor.java

Rules for recursive-descent parsers

- Each nonterminal becomes a (void) procedure.
- RHS symbols become statements.
  - *Terminals* become "match" statements.
  - *Nonterminals* become procedure calls.
  - *Symbol sequences* become statement sequences.
  - *Star/plus repetitions* become while loops with lookahead.
  - *Alternatives* are selected based on lookahead.

Builds on top of
the *Acceptor Pattern*

The *Parser* Pattern

The *Parser* Pattern

- Intent:

  Make accessible syntactical structure.

- Operational steps (run time):
  - ✦ Accept input based on the *Acceptor Pattern*.
  - ✦ Invoke semantic actions along acceptance.

For comparison: **no** semantic actions

```
void department() {
    match(DEPARTMENT);
    match(STRING);
    match(OPEN);
    match(MANAGER);
    employee();
    while (test(EMPLOYEE)) {
        match(EMPLOYEE);
        employee();
    }
    while (test(DEPARTMENT))
        department();
    match(CLOSE);
}
```


```
void department() {
    match(DEPARTMENT);
    String name = match(STRING);
    match(OPEN);
    openDept(name);
    match(MANAGER);
    employee(true);
    while (test(EMPLOYEE)) {
        match(EMPLOYEE);
        employee(false);
    }
    while (test(DEPARTMENT))
        department();
    match(CLOSE);
    closeDept(name);
}
```

All handlers for companies

```
protected void openCompany(String name) { }
protected void closeCompany(String name) { }
protected void openDept(String name) { }
protected void closeDept(String name) { }
protected void handleEmployee(
        boolean isManager,
        String name,
        String address,
        Double salary) { }
```

A parser that totals

```
public class Total extends Parser {

    private double total = 0;

    public double getTotal() {
        return total;
    }
    protected void handleEmployee(
                boolean isFinal,
                String name,
                String address,
                Double salary) {
        total += salary;
    }
}
```

Demo

http://101companies.org/wiki/Contribution:**javaParser**

See class Parser.java

Summary

- Language processing is a programming domain.
- Grammars may define two levels of syntax:
  - token/lexeme level (lexical level)
  - tree-like structure level (context-free level)
- Both levels are implementable in parsers:
  - Recursive descent for parsing
  - Use generative tools (as discussed soon)