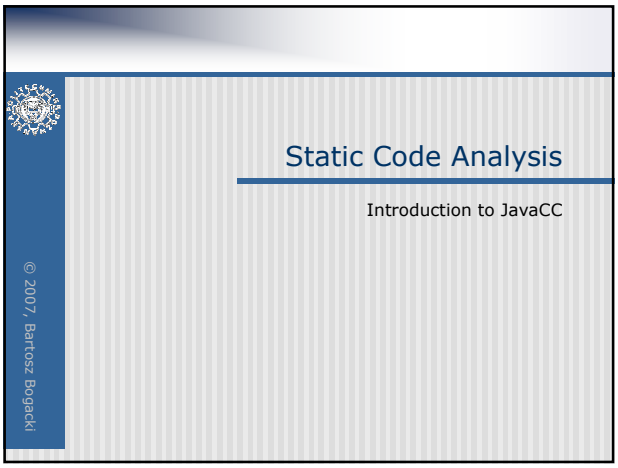


JavaCC features

- JavaCC is a Java parser generator written in Java programming language
- It produces pure Java code
- JavaCC generates top-down (*recursive descent*) parsers as opposed to bottom-up parsers generated by YACC-like tools

JavaCC – specification file

- Specification file has *.jj extension
- Lexical and grammar specification is in one file
- The file has the following format:

javacc options
PARSER_BEGIN (<identifier>**)**
java compilation unit
PARSER_END (<identifier>**)**
production rules

<identifier> stands for parser name and should be the same as *.jj filename.

JavaCC options

- Sample *javacc options* section

```
options {
    LOOKAHEAD = 1;
    CHOICE_AMBIGUITY_CHECK = 2;
    OTHER_AMBIGUITY_CHECK = 1;
    STATIC = true;
    DEBUG_PARSER = false;
    DEBUG_LOOKAHEAD = false;
    DEBUG_TOKEN_MANAGER = false;
    ERROR_REPORTING = true;
    JAVA_UNICODE_ESCAPE = false;
    UNICODE_INPUT = false;
    IGNORE_CASE = false;
    USER_TOKEN_MANAGER = false;
    USER_CHAR_STREAM = false;
    BUILD_PARSER = true;
    BUILD_TOKEN_MANAGER = true;
    SANITY_CHECK = true;
    FORCE_LA_CHECK = false;
}
```


JavaCC compilation unit

- Sample *java compilation unit* section

```
import java.io.StringBufferInputStream;

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

    ParserExample parser =
      new ParserExample(
        new StringBufferInputStream ("abbba"));
    parser.start()
  }
}
```

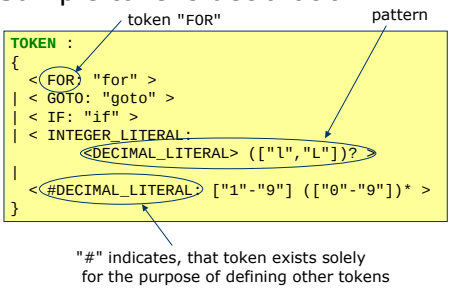
non-terminal marked as start symbol

JavaCC production rules

- *Production rules* section consist of
  - Tokens specification
  - Grammar rules specification

JavaCC tokens specification

- Sample tokens declaration



JavaCC tokens specification

- Regular expressions in JavaCC
  - ("ab"|"c") — ab, c
  - (("ab")*"c") — c, abc, ababc
  - (("ab")+"c") — abc, ababc, abababc
  - (("ab")?"c") — c, abc
  - ["a"-"e"] — a, b, c, d, e
  - ["a"-"c"]["1"-"3"] — a1, a2, a3, b1, b2,...
  - ~["b","c","d","g"] — a, e, f, h, i, j, k,...
  - ~[] — *any character (.)*

JavaCC tokens specification

- JavaCC provides 4 kinds of regular expression specification:
  - TOKEN
  - SPECIAL_TOKEN – tokens ignored by grammar rules, but accessible in parser
  - SKIP – to ignore some input
  - MORE – to build tokens in several steps

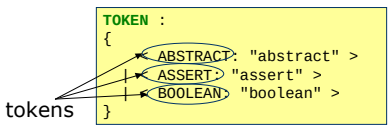
JavaCC tokens specification

- SKIP kind

```
SKIP :
{
  " "
| "\t"
| "\n"
}
```


JavaCC tokens specification

- TOKEN kind



JavaCC tokens specification

- MORE kind

```
MORE :
{
  "in" : REST
}

<REST>
TOKEN :
{
   < INSTANCEOF: "stanceof" >
 | < INT: "t" >
 | < INTERFACE: "terface" >
}
```

JavaCC tokens specification

- JavaCC allows for *lexical states*

```
MORE :
{
  "//" : IN_SINGLE_LINE_COMMENT
}

<IN_SINGLE_LINE_COMMENT>
SPECIAL_TOKEN :
{
  <SINGLE_LINE_COMMENT: "\n" | "\r" | "\r\n" > : DEFAULT
}
```

start state: IN_SINGLE_LINE_COMMENT

execute only in state: IN_SINGLE_LINE_COMMENT

start state: DEFAULT

JavaCC tokens specification

- JavaCC allows for execution of *actions* which are bound to regexps

```
SKIP :
{
    "\n" { line++; }
}
```

action

- All variables used during lexical analysis can be declared in TOKEN_MGR_DECLS section, i.e.:

```
TOKEN_MGR_DECLS :
{
  private static int line = 0;
}
```


JavaCC tokens specification

- Variables available for use within lexical actions:
  - StringBuffer **image**
  - int **lengthOfMatch**
  - int **curLexState**
  - (ASCII_CharStream |ASCII_UCodeESC_CharStream |UCode_CharStream |UCode_UCodeESC_CharStream) **inputStream**
  - Token **matchedToken**

JavaCC tokens specification

- **image** contains all the characters that have been matched since the last:
  - SKIP
  - TOKEN
  - SPECIAL_TOKEN
- **lengthOfMatch** is the length of the current match (is not cumulative over MORE's)

JavaCC tokens specification

- Example:

```
TOKEN_MGR_DECLS :
{
  private static int forCounter = 0;
}

TOKEN :
{
  < FOR: "for" > {forCounter++;} : IN_FOR
}

<IN_FOR>
SKIP :
{
  " " : DEFAULT
}
```


JavaCC grammar rules specification

- JavaCC specification for the following grammar:

```
A -> a B a
B -> b
B -> c
```

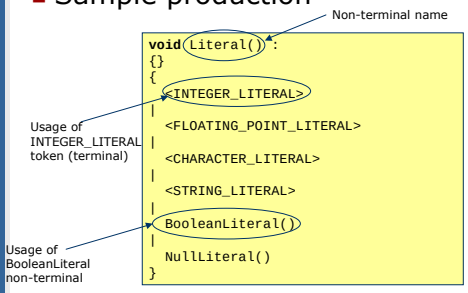
- is:

```
void A() :
{}
{
  "a" B() "a"
}

void B() :
{}
{
  "b"
  |
  "c"
}
```

JavaCC grammar rules specification

- Sample production

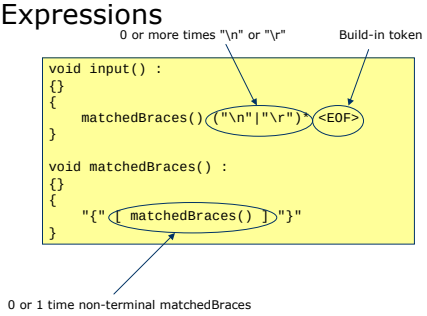


JavaCC grammar rules specification

- Some additional expressions can be used in productions according to BNF (Backus-Naur Form), like:
  - [NT()] 0 or 1 non-terminal NT
  - ["x"] 0 or 1 terminal "x"
  - (NT())? 0 or 1 non-terminal NT
  - ("x")? 0 or 1 terminal "x"
  - (NT())* 0 or more non-terminals NT
  - ("x")* 0 or more terminals "x"
  - (NT())+ 1 or more non-terminals NT
  - ("x")+ 1 or more terminals "x"

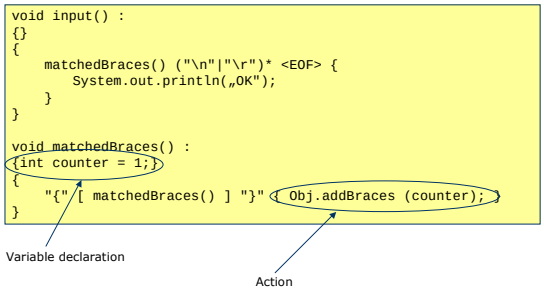
JavaCC grammar rules specification

- Expressions



JavaCC grammar rules specification

- Actions



JavaCC grammar rules specification

- Access to the lexems

```
TOKEN :
{
  < NUMBER: (["0"-"9"])+ >
}

void NT() :
{Token token;}
{
   token=<NUMBER> X() <EOF> {
        System.out.println("Lexem is: " + token.image);
    }
}
```


JavaCC grammar rules specification

- Class Token by default includes the following public fields:
  - `String image`
  - `int kind`
  - `int beginLine`
  - `int beginColumn`
  - `int endLine`
  - `int endColumn`
  - `Token next`
  - `Token specialToken`

} To access **SPECIAL_TOKEN**s

JavaCC grammar rules specification

- Attributes

```
TOKEN:
{
  <INT: "int">
  | <FLOAT: "float">
  | <ID: (["a"-"z"])+ >
}
```

```
void s() : {String name; Token t;}
{
  (t=<INT>|t=<FLOAT>) name=v(t.image) <EOF> {
        System.out.println (name);
  }
}

String v(String type) : {Token t;}
{
  t=<ID> {
    if ("int".equals(type)) return t.image;
    else return "NOT INT!";
  }
}
```


JAVACODE

- Sometimes plain java code has to be mixed with standard EBNFs
- Example – nested comments:

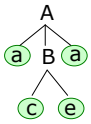
```
JAVACODE
void skip_to_matching_close_comment() {
  Token tok;
  int nesting = 1;
  while (true) {
    tok = getToken(1);
    if (tok.kind == OPEN_COMMENT) nesting++;
    if (tok.kind == CLOSE_COMMENT) {
      --nesting;
      if (nesting == 0) break;
    }
    tok = getNextToken();
  }
}
```


JavaCC grammar rules specification

- JavaCC generates predictive parsers that "look ahead" for tokens from input stream to decide which production to use
- By default it generates parser that checks for 1 token ahead - LL(1) grammar
- Consider the following example:

```
1) A -> a B a
2) B -> b c d
3) B -> c e
```

Input stream:



JavaCC grammar rules specification

- ...but what if our grammar is

```
1) A -> a B a
2) B -> b c d
3) B -> b e
```

Input stream: **a b e a**

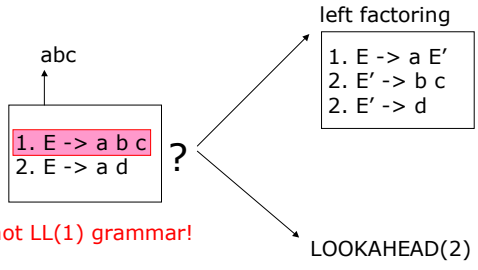
We have to perform *left factoring...*
...but in some cases, this may lead to overcomplicated grammar.

If so, then we can shift from LL(1) to LL(n) by specifing greater number of "look ahead".

LOOKAHEAD

- The number of tokens to look ahead before making a decision at a choice point during parsing.
- For LL(1) parsers the lookahead is 1
- For efficiency reasons, it should be set to look for 1 symbol only
- It should be switched for greater values locally

LOOKAHEAD



LOOKAHEAD

- Lookahead value can be specified either for whole grammar (in *javacc_options* section) or locally for concrete production
- There are 3 types of LOOKAHEADs:
  - Lookahead Limit
  - Syntactic Lookahead
  - Semantic Lookahead

LOOKAHEAD

- Lookahead Limit

```
void B() : {}
{
   LOOKAHEAD(2)
   "b" "c" "d"
   |
   "b" "e"
}
```

- Syntactic Lookahead

```
void TypeDeclaration() : {}
{
   LOOKAHEAD(ClassDeclaration())
   ClassDeclaration()
   |
   InterfaceDeclaration()
}
```

Both `ClassDeclaration` and `InterfaceDeclaration` can start with any number of "`abstract`", "`final`", and "`public`" modifiers.

LOOKAHEAD

- Semantic Lookahead

```
void A() :
{}
{
(
   LOOKAHEAD({getToken(1).kind == GT})
   ">" B() )
}
```


LOOKAHEAD

- LOOKAHEAD limit can be modified globally in *options* section

```
options {
    LOOKAHEAD = 2;
}
```


Static methods

- By default, all methods and fields generated by JavaCC have *static* specifier
- This behaviour can be changed by setting STATIC option to false

```
options {
    STATIC = false;
}
```


Debugging

- The parser may have debugging mode enabled with DEBUG_PARSER option
- enable_tracing() / disable_tracing()

```
options {
    DEBUG_PARSER = true;
}
```

Debugging

```
void S() : {}
{
L() <EOF>
}

void L() : {}
{
"a" M()
}

void M() : {}
{
"b"
|
L() "b"
}
```

```
Call:   S
  Call:   L
    Consumed token: <"a" at line 1 column 1>
    Call:   M
      Call:   L
        Consumed token: <"a" at line 1 column 2>
        Call:   M
          Call:   L
            Consumed token: <"a" at line 1 column 3>
            Call:   M
              Consumed token: <"b" at line 1 column 4>
            Return: M
          Return: L
          Consumed token: <"b" at line 1 column 5>
        Return: M
      Return: L
      Consumed token: <"b" at line 1 column 6>
    Return: M
  Return: L
  Consumed token: <<EOF> at line 1 column 6>
Return: S
```


Debugging

- The scanner may have debugging mode enabled with DEBUG_TOKEN_MANAGER option

```
options {
    DEBUG_TOKEN_MANAGER = true;
}
```


Debugging

aaa

```
Current character : a (97) at line 1 column 1
   No more string literal token matches are possible.
   Currently matched the first 1 characters as a "a" token.
****** FOUND A "a" MATCH (a) ******

Current character : a (97) at line 1 column 2
   No more string literal token matches are possible.
   Currently matched the first 1 characters as a "a" token.
****** FOUND A "a" MATCH (a) ******

Current character : a (97) at line 1 column 3
   No more string literal token matches are possible.
   Currently matched the first 1 characters as a "a" token.
****** FOUND A "a" MATCH (a) ******

Returning the <EOF> token.
```


Unicode

- By default JavaCC assumes ASCII files as an input
- To enable the generated parser to use Unicode files, UNICODE_INPUT option should be set to true
- To enable the generated parser to proces Java Unicode escapes (\u...), JAVA_UNICODE_ESCAPE option should be set to true

```
options {
    UNICODE_INPUT = true;
    JAVA_UNICODE_ESCAPE = true;
}
```


Case sensitivity

- By default JavaCC generates scanner that is case-sensitive
- This behaviour can be modified with IGNORE_CASE option

```
options {
    IGNORE_CASE = true;
}
```


User defined scanner

- User defined scanner can be used for parser generated with JavaCC options:
  - **USER_TOKEN_MANAGER**
  - **BUILD_TOKEN_MANAGER**
- Parser generation can be disabled with option **BUILD_PARSER** (only scanner is built)

Error recovery

- There are 2 types of error recovery
- Shallow recovery
  - if none of the current choices have succeeded in being selected
- Deep recovery
  - if a choice was selected, but then an error happens sometime during the parsing of this choice

Shallow recovery

- New Non-Terminal error_skipto() is added
- Recovery is done only if an error occures in Non-Terminal A

```
void A() :
{}
{
    B()
  | C()
  | error_skipto(SOME_TERMINAL)
}
```


Shallow recovery

- Sample implementation

```
JAVACODE
void error_skipto(int kind) {
  ParseException e = generateParseException();
  System.out.println(e.toString());
  Token t;
  do {
    t = getNextToken();
  } while (t.kind != kind);
}
```


Deep recovery

- Recovers even if an error occures within B or C

```
void A() :
{}
{
  try {
    (
      B()
    | C()
    )
  } catch (ParseException e) {
    System.out.println(e.toString());
    Token t;
    do {
      t = getNextToken();
    } while (t.kind != SOME_TERMINAL);
  }
}
```


JavaCC execution

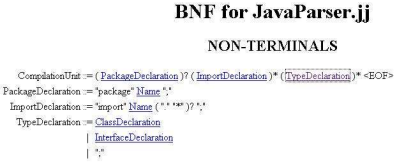
- As a result of execution JavaCC compiler on the specification file, the following files should appear:
  - <ParserName>.java (The parser)
  - <ParserName>TokenManager.java (The scanner)
  - <ParserName>Constants.java (Constants)

  where <ParserName> is the name specified in PARSER_BEGIN/PARSER_END
- JavaCC can be run as ***ant*** task:

```
<javacc target="parser.jj"
  outputdirectory="${output.dir}"
  javacchome="${javacc.home}"/>
```


JavaCC extensions

- There are number of extensions for JavaCC, like:
  - JJTree – for creation of AST
  - JJDoc – for HTML documentation of BNF



BNF for JavaParser.jj

NON-TERMINALS

CompilationUnit ::= (PackageDeclaration)? (ImportDeclaration)* (TypeDeclaration)* <EOF>
PackageDeclaration ::= "package" Name ";"
ImportDeclaration ::= "import" Name ("." "*")? ";"
TypeDeclaration ::= ClassDeclaration
| InterfaceDeclaration
| ";"

The end...

Thank you for your attention!