Introduction to Compiler Construction

Scanning: JavaCC Scanner for *j--*

Outline

1. JavaCC Overview
2. Scanning in JavaCC
3. JavaCC Scanner for *j--*

JavaCC Overview

JavaCC Overview

JavaCC is a tool for generating scanners from lexical grammars and parsers from syntactic grammars

JavaCC Overview

JavaCC is a tool for generating scanners from lexical grammars and parsers from syntactic grammars

The lexical and the syntactic grammars are both included within the same input file having a `.jj` extension

JavaCC Overview

JavaCC is a tool for generating scanners from lexical grammars and parsers from syntactic grammars

The lexical and the syntactic grammars are both included within the same input file having a `.jj` extension

JavaCC allows BNF syntax such as `( A )*` within the lexical and syntactic grammars

Scanning in JavaCC

Scanning in JavaCC

A lexical grammar consists a set of regular expressions and a set of lexical states

Scanning in JavaCC

A lexical grammar consists a set of regular expressions and a set of lexical states

Scanning begins in the `DEFAULT` state

Scanning in JavaCC

A lexical grammar consists a set of regular expressions and a set of lexical states

Scanning begins in the `DEFAULT` state

From a particular state, a set of regular expressions may be matched by the input

Scanning in JavaCC

A lexical grammar consists a set of regular expressions and a set of lexical states

Scanning begins in the `DEFAULT` state

From a particular state, a set of regular expressions may be matched by the input

From the matched regular expressions, the scanner picks the one that consumes the most number of input characters

Scanning in JavaCC

A lexical grammar consists a set of regular expressions and a set of lexical states

Scanning begins in the `DEFAULT` state

From a particular state, a set of regular expressions may be matched by the input

From the matched regular expressions, the scanner picks the one that consumes the most number of input characters

After a match, the scanner may transition into a different state or stay in the current state

Scanning in JavaCC

Scanning in JavaCC

Four types of regular expressions

1. `MORE`: continues to the next state, taking the matched string along
2. `SKIP`: throws away the matched string
3. `SPECIAL_TOKEN`: creates a special token that does not participate in the parsing
4. `TOKEN`: creates a token from the matched string and returns it to the parser

Scanning in JavaCC

Scanning in JavaCC

BNF syntax

- `( a )?` for "zero or one" occurrence of `a`
- `( a )*` for "zero or more" occurrences of `a`
- `( a | b )` for either `a` or `b`
- `[ "a" - "d", "x", "y"]` for `a`, `b`, `c`, `d`, `x`, or `y`
- `( )` for grouping

JavaCC Scanner for j--

JavaCC Scanner for j--

JavaCC generates a scanner for *j--* from regular expressions defined in `$j/j--/src/jminusminus/j--.jj`

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

Method 1

```
SKIP: { <BEGIN_COMMENT: "//">: IN_SINGLE_LINE_COMMENT }
<IN_SINGLE_LINE_COMMENT>
SKIP: { <END_COMMENT: "\n" | "\r" | "\r\n">: DEFAULT }
<IN_SINGLE_LINE_COMMENT>
SKIP: { <COMMENT: ~[]> }
```

Method 1

```
SKIP: { <BEGIN_COMMENT: "//">: IN_SINGLE_LINE_COMMENT }
<IN_SINGLE_LINE_COMMENT>
SKIP: { <END_COMMENT: "\n" | "\r" | "\r\n">: DEFAULT }
<IN_SINGLE_LINE_COMMENT>
SKIP: { <COMMENT: ~[]> }
```

Method 2

```
SPECIAL_TOKEN: {
  <SINGLE_LINE_COMMENT: "//" ( ~[ "\n", "\r" ] )* ( "\n" | "\r" | "\r\n" )>
}
```

Reserved words

```
TOKEN: {
  <ABSTRACT: "abstract">
| <BOOLEAN: "boolean">
...
| <WHILE: "while">
}
```

Reserved words

```
TOKEN: {
  <ABSTRACT: "abstract">
| <BOOLEAN: "boolean">
...
| <WHILE: "while">
}
```

Separators

```
TOKEN: {
  <COMMA: ",">
| <DOT: ".">
...
| <SEMI: ";">
}
```

Reserved words

```
TOKEN: {
  <ABSTRACT: "abstract">
| <BOOLEAN: "boolean">
...
| <WHILE: "while">
}
```

Separators

```
TOKEN: {
  <COMMA: ",">
| <DOT: ".">
...
| <SEMI: ";">
}
```

Operators

```
TOKEN: {
  <ASSIGN: "=">
| <DEC: "--">
...
| <STAR: "*">
}
```

```
TOKEN: {
  <IDENTIFIER: ( <LETTER> | "_" | "$" ) ( <LETTER> | <DIGIT> | "_" | "$" )*>
| <#LETTER: [ "a"-"z", "A"-"Z" ]>
| <#DIGIT: [ "0"-"9" ]>
}
```

```
TOKEN: {
  <INT_LITERAL: <DIGITS>>
| <CHAR_LITERAL: "'" ( <ESC> | ~[ "'", "\\" ] ) "'">
| <STRING_LITERAL: "\"" ( <ESC> | ~[ "\"", "\\" ] )* "\"">
| <#DIGITS: <DIGIT> ( <DIGIT> )* >
| <#ESC: "\\" [ "n", "t", "b", "r", "f", "\\", "'", "\"" ]>
}
```