Introduction to Compiler Construction

Parsing: JavaCC Parser for *j--*

Outline

1. Overview
2. Parsing in JavaCC
3. JavaCC Parser for *j--*

Overview

Overview

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

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

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

Parsing in JavaCC

Parsing in JavaCC

JavaCC generates an LL(1) parser

Parsing in JavaCC

JavaCC generates an LL(1) parser

For portions of the grammar that are not LL(1), JavaCC offers a lookahead feature to resolve local ambiguities

Parsing in JavaCC

JavaCC generates an LL(1) parser

For portions of the grammar that are not LL(1), JavaCC offers a lookahead feature to resolve local ambiguities

The Java code between the `PARSER_BEGIN(XYZ)` and `PARSER_END(XYZ)` block in the input `.jj` file is copied verbatim into the generated `XYZ.java` file

Parsing in JavaCC

JavaCC generates an LL(1) parser

For portions of the grammar that are not LL(1), JavaCC offers a lookahead feature to resolve local ambiguities

The Java code between the `PARSER_BEGIN(XYZ)` and `PARSER_END(XYZ)` block in the input `.jj` file is copied verbatim into the generated `XYZ.java` file

Following the block is the specification for the scanner and following that is the specification for the parser

Parsing in JavaCC

JavaCC generates an LL(1) parser

For portions of the grammar that are not LL(1), JavaCC offers a lookahead feature to resolve local ambiguities

The Java code between the `PARSER_BEGIN(XYZ)` and `PARSER_END(XYZ)` block in the input `.jj` file is copied verbatim into the generated `XYZ.java` file

Following the block is the specification for the scanner and following that is the specification for the parser

The input file defines a start symbol, which is a high level non-terminal that references other lower level non-terminals, which in turn reference the tokens

Parsing in JavaCC

Parsing in JavaCC

Syntax for a non-terminal declaration:

</> j--.jj

```
private|public <type> <name>(<parameter1>, <parameter2>, ...):
{
    // Local variables.
    ...
}
{
    try {
      // Grammar rules along with syntactic actions.
      ...
    } catch (ParseException e) {
        recoverFromError(new int[] { SEMI, EOF }, e);
    }
    {
        return <expression>;
    }
}
```

Parsing in JavaCC

Syntax for a non-terminal declaration:

</> j--.jj

```
private|public <type> <name>(<parameter1>, <parameter2>, ...):
{
    // Local variables.
    ...
}
{
    try {
      // Grammar rules along with syntactic actions.
      ...
    } catch (ParseException e) {
        recoverFromError(new int[] { SEMI, EOF }, e);
    }
    {
        return <expression>;
    }
}
```

Syntactic actions, such as creating/returning an AST node, are Java statements embedded within curly braces

Parsing in JavaCC

Syntax for a non-terminal declaration:

</> j--.jj

```
private|public <type> <name>(<parameter1>, <parameter2>, ...):
{
    // Local variables.
    ...
}
{
    try {
      // Grammar rules along with syntactic actions.
      ...
    } catch (ParseException e) {
        recoverFromError(new int[] { SEMI, EOF }, e);
    }
    {
        return <expression>;
    }
}
```

Syntactic actions, such as creating/returning an AST node, are Java statements embedded within curly braces

JavaCC turns the specification for each non-terminal into a Java method within the generated parser

Parsing in JavaCC

Parsing 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`
- `( )` for grouping

JavaCC Parser for j--

JavaCC Parser for j--

JavaCC generates a parser for *j--* from context-free grammar rules (and the corresponding syntactic actions) defined in `$j/j--/src/jminusminus/j--.jj`:

```
compilationUnit ::= [ PACKAGE qualifiedIdentifier SEMI ]
                    { IMPORT  qualifiedIdentifier SEMI }
                    { typeDeclaration }
                    EOF
```

</> j--.jj 1/2

```
public JCompilationUnit compilationUnit():
{
    int line = 0;
    TypeName packageName = null;
    TypeName anImport = null;
    ArrayList<TypeName> imports = new ArrayList<>();
    JAST aTypeDeclaration = null;
    ArrayList<JAST> typeDeclarations = new ArrayList<>();
}
{
    try {
        [
            <PACKAGE>
            { line = token.beginLine; }
            packageName = qualifiedIdentifier()
            <SEMI>
        ]
        (
            <IMPORT>
            { line = line == 0 ? token.beginLine : line; }
            anImport = qualifiedIdentifier()
            { imports.add(anImport); }
            <SEMI>
        )*
        (
            aTypeDeclaration = typeDeclaration()
            {
                line = line == 0 ? aTypeDeclaration.line() : line;
                typeDeclarations.add(aTypeDeclaration);
            }
        )*
        <EOF>
        { line = line == 0 ? token.beginLine : line; }
    } catch (ParseException e) {
        recoverFromError(new int[] { SEMI, EOF }, e);
```

</> j--.jj 2/2

```
36      }
37      { return new JCompilationUnit(fileName, line, packageName, imports, typeDeclarations); }
38  }
```

```
qualifiedIdentifier ::= IDENTIFIER { DOT IDENTIFIER }
```

```
qualifiedIdentifier ::= IDENTIFIER { DOT IDENTIFIER }
```

</> j--.jj

```
1  private TypeName qualifiedIdentifier():
2  {
3      int line = 0;
4      String qualifiedIdentifier = "";
5  }
6  {
7      try {
8          <IDENTIFIER>
9          {
10             line = token.beginLine;
11             qualifiedIdentifier = token.image;
12         }
13         (
14             LOOKAHEAD(<DOT> <IDENTIFIER>)
15             <DOT> <IDENTIFIER>
16             { qualifiedIdentifier += "." + token.image; }
17         )*
18     } catch (ParseException e) {
19         recoverFromError(new int[] { SEMI, EOF }, e);
20     }
21     { return new TypeName(line, qualifiedIdentifier); }
22 }
```

```
statement ::= block
            | IF parExpression statement [ ELSE statement ]
            | RETURN [ expression ] SEMI
            | SEMI
            | WHILE parExpression statement
            | statementExpression SEMI
```

</> j--.jj 1/2

```
1  private JStatement statement():
2  {
3      int line = 0;
4      JExpression expr      = null;
5      JStatement statement  = null;
6      JStatement consequent = null;
7      JStatement alternate  = null;
8      JStatement body       = null;
9  }
10 {
11     try {
12         statement = block() |
13         <IF>
14         { line = token.beginLine; }
15         expr = parExpression()
16         consequent = statement()
17         [
18             LOOKAHEAD(<ELSE>)
19             <ELSE>
20             alternate = statement()
21         ]
22         { statement = new JIfStatement(line, expr, consequent, alternate); } |
23         <RETURN>
24         { line = token.beginLine; }
25         [
26             expr = expression()
27         ]
28         <SEMI>
29         { statement = new JReturnStatement(line, expr); } |
30         <SEMI>
31         {
32             line = token.beginLine;
33             statement = new JEmptyStatement( line );
34         } |
35         <WHILE>
```

j--.jj 2/2

```
36          { line = token.beginLine; }
37          expr = parExpression()
38          body = statement()
39          { statement = new JWhileStatement(line, expr, body); } |
40          statement = statementExpression()
41          <SEMI>
42      } catch (ParseException e) {
43          recoverFromError(new int[] { SEMI, EOF }, e);
44      }
45      { return statement; }
46  }
```

```
simpleUnaryExpression ::= LNOT unaryExpression
                        | LPAREN basicType RPAREN unaryExpression
                        | LPAREN referenceType RPAREN simpleUnaryExpression
                        | postfixExpression
```

</> j--.jj

```
private JExpression simpleUnaryExpression():
{
    int line = 0;
    Type type = null;
    JExpression expr = null, unaryExpr = null, simpleUnaryExpr = null;
}
{
    try {
        <LNOT>
        { line = token.beginLine; }
        unaryExpr = unaryExpression()
        { expr = new JLogicalNotOp(line, unaryExpr); } |
        LOOKAHEAD(<LPAREN> basicType() <RPAREN>)
        <LPAREN>
        { line = token.beginLine; }
        type = basicType()
        <RPAREN>
        unaryExpr = unaryExpression()
        { expr = new JCastOp(line, type, unaryExpr); } |
        LOOKAHEAD(<LPAREN> referenceType() <RPAREN>)
        <LPAREN>
        { line = token.beginLine; }
        type = referenceType()
        <RPAREN>
        simpleUnaryExpr = simpleUnaryExpression()
        { expr = new JCastOp(line, type, simpleUnaryExpr); } |
        expr = postfixExpression()
    } catch (ParseException e) {
        recoverFromError(new int[] { SEMI, EOF }, e);
    }
    { return expr ; }
}
```

Our error recovery scheme involves catching the `ParseException` instance that is raised in the event of a syntax error

Our error recovery scheme involves catching the `ParseException` instance that is raised in the event of a syntax error

The exception instance `e` and an array of tokens `skipTo` are passed to the `recoverFromError()` method

- Our error recovery scheme involves catching the `ParseException` instance that is raised in the event of a syntax error
- The exception instance `e` and an array of tokens `skipTo` are passed to the `recoverFromError()` method
- The exception instance has information about the token that was found and the token that was expected

- Our error recovery scheme involves catching the `ParseException` instance that is raised in the event of a syntax error
- The exception instance `e` and an array of tokens `skipTo` are passed to the `recoverFromError()` method
- The exception instance has information about the token that was found and the token that was expected
- The array has tokens (`SEMI` and `EOF`) to skip to in order to recover from the error

Our error recovery scheme involves catching the `ParseException` instance that is raised in the event of a syntax error

The exception instance `e` and an array of tokens `skipTo` are passed to the `recoverFromError()` method

The exception instance has information about the token that was found and the token that was expected

The array has tokens (`SEMI` and `EOF`) to skip to in order to recover from the error

When `ParseException` is raised, control is transferred to the calling non-terminal, leaving the lower non-terminals unparsed

</> j--.jj

```
    private void recoverFromError(int[] skipTo, ParseException e) {
        StringBuffer expected = new StringBuffer();
        for (int i = 0; i < e.expectedTokenSequences.length; i++) {
            for (int j = 0; j < e.expectedTokenSequences[i].length; j++) {
                expected.append("\n");
                expected.append("    ");
                expected.append(tokenImage[e.expectedTokenSequences[i][j]]);
                expected.append("...");
            }
        }

        if (e.expectedTokenSequences.length == 1) {
            reportParserError("\"%s\" found where %s sought", getToken(1), expected);
        } else {
            reportParserError("\"%s\" found where one of %s sought", getToken(1), expected);
        }

        boolean loop = true;
        do {
            token = getNextToken();
            for (int i = 0; i < skipTo.length; i++) {
                if (token.kind == skipTo[i]) {
                    loop = false;
                    break;
                }
            }
        } while(loop);
    }
```