Contents | Prev | Next | Index

CHAPTER 19

LALR(1) Grammar

This chapter presents a grammar for Java. The grammar has been mechanically checked to insure that it is LALR(1).

The grammar for Java presented piecemeal in the preceding chapters is much better for exposition, but it cannot be parsed left-to-right with one token of lookahead because of certain syntactic peculiarities, some of them inherited from C and C++. These problems and the solutions adopted for the LALR(1) grammar are presented below, followed by the grammar itself.

19.1 Grammatical Difficulties

There are five problems with the grammar presented in preceding chapters.

19.1.1 Problem #1: Names Too Specific

Consider the two groups of productions:

```
PackageName:
        Identifier
        PackageName . Identifier
TypeName:
        Identifier
        PackageName . Identifier
```

and:

```
MethodName:
        Identifier
        AmbiguousName . Identifier
AmbiguousName:
        Identifier
        AmbiguousName . Identifier
```

Now consider the partial input:

```
class Problem1 { int m() { hayden.
```

When the parser is considering the token `hayden`, with one-token lookahead to symbol ".", it cannot yet tell whether `hayden` should be a *PackageName* that qualifies a type name, as in:

```
hayden.Dinosaur rex = new hayden.Dinosaur(2);
```

or an *AmbiguousName* that qualifies a method name, as in:

```
hayden.print("Dinosaur Rex!");
```

Therefore, the productions shown above result in a grammar that is not LALR(1). There are also other problems with drawing distinctions among different kinds of names in the grammar.

The solution is to eliminate the nonterminals *PackageName*, *TypeName*, *ExpressionName*, *MethodName*, and *AmbiguousName*, replacing them all with a single nonterminal *Name*:

```
Name:

        SimpleName

        QualifiedName

SimpleName:

        Identifier

QualifiedName:

        Name . Identifier
```

A later stage of compiler analysis then sorts out the precise role of each name or name qualifier.

For related reasons, these productions in §4.3:

```
ClassOrInterfaceType:

        ClassType

        InterfaceType

ClassType:

        TypeName

InterfaceType:

        TypeName
```

were changed to:

```
ClassOrInterfaceType:

        Name

ClassType:

        ClassOrInterfaceType

InterfaceType:

        ClassOrInterfaceType
```

19.1.2 Problem #2: Modifiers Too Specific

Consider the two groups of productions:

FieldDeclaration:

FieldModifiers$_{opt}$ *Type VariableDeclarators* ;

FieldModifiers:

FieldModifier

FieldModifiers FieldModifier

FieldModifier: one of

`public protected private`

`final static transient volatile`

and:

MethodHeader:

MethodModifiers$_{opt}$ *ResultType MethodDeclarator Throws*$_{opt}$

MethodModifiers:

MethodModifier

MethodModifiers MethodModifier

MethodModifier: one of

`public protected private`

`static`

`abstract final native synchronized`

Now consider the partial input:

```
class Problem2 { public static int
```

When the parser is considering the token `static`, with one-token lookahead to symbol `int`-or, worse yet, considering the token `public` with lookahead to `static`-it cannot yet tell whether this will be a field declaration such as:

```
public static int maddie = 0;
```

or a method declaration such as:

```
public static int maddie(String art) { return art.length(); }
```

Therefore, the parser cannot tell with only one-token lookahead whether `static` (or, similarly, `public`) should be reduced to *FieldModifier* or *MethodModifier*. Therefore, the productions shown above result in a grammar that is not LALR(1). There are also other problems with drawing distinctions among different kinds of modifiers in the grammar.

While not all contexts provoke the problem, the simplest solution is to combine all contexts in which such modifiers are used, eliminating all six of the nonterminals *ClassModifiers* (§8.1.2), *FieldModifiers* (§8.3.1), *MethodModifiers* (§8.4.3), *ConstructorModifiers* (§8.6.3), *InterfaceModifiers* (§9.1.2), and *ConstantModifiers* (§9.3) from the grammar, replacing them all with a single nonterminal *Modifiers*:

Modifiers:

Modifier

Modifiers Modifier

Modifier: one of

`public protected private`

`static`

`abstract final native synchronized transient volatile`

A later stage of compiler analysis then sorts out the precise role of each modifier and whether it is permitted in a given context.

19.1.3 Problem #3: Field Declaration versus Method Declaration

Consider the two productions (shown after problem #2 has been corrected):

FieldDeclaration:

Modifiers$_{opt}$ *Type VariableDeclarators* ;

and:

MethodHeader:

Modifiers$_{opt}$ *ResultType MethodDeclarator Throws*$_{opt}$

where *ResultType* is defined as:

ResultType:

Type

`void`

Now consider the partial input:

```
class Problem3 { int julie
```

Note that, in this simple example, no *Modifiers* are present. When the parser is considering the token `int`, with one-token lookahead to symbol `julie`, it cannot yet tell whether this will be a field declaration such as:

```
int julie = 14;
```

or a method declaration such as:

```
int julie(String art) { return art.length(); }
```

Therefore, after the parser reduces `int` to the nonterminal *Type*, it cannot tell with only one-token lookahead whether *Type* should be further reduced to *ResultType* (for a method declaration) or left alone (for a field declaration). Therefore, the productions shown above result in a grammar that is not LALR(1).

The solution is to eliminate the *ResultType* production and to have separate alternatives for *MethodHeader*:

MethodHeader:

Modifiers$_{opt}$ *Type MethodDeclarator Throws*$_{opt}$

Modifiers$_{opt}$ `void` *MethodDeclarator* *Throws*$_{opt}$

This allows the parser to reduce `int` to *Type* and then leave it as is, delaying the decision as to whether a field declaration or method declaration is in progress.

19.1.4 Problem #4: Array Type versus Array Access

Consider the productions (shown after problem #1 has been corrected):

ArrayType:

Type `[ ]`

and:

ArrayAccess:

Name `[` *Expression* `]`

PrimaryNoNewArray `[` *Expression* `]`

Now consider the partial input:

```
class Problem4 { Problem4() { peter[
```

When the parser is considering the token `peter`, with one-token lookahead to symbol `[`, it cannot yet tell whether `peter` will be part of a type name, as in:

```
peter[] team;
```

or part of an array access, as in:

```
peter[3] = 12;
```

Therefore, after the parser reduces `peter` to the nonterminal *Name*, it cannot tell with only one-token lookahead whether *Name* should be reduced ultimately to *Type* (for an array type) or left alone (for an array access). Therefore, the productions shown above result in a grammar that is not LALR(1).

The solution is to have separate alternatives for *ArrayType*:

ArrayType:

PrimitiveType `[ ]`

Name `[ ]`

ArrayType `[ ]`

This allows the parser to reduce `peter` to *Name* and then leave it as is, delaying the decision as to whether an array type or array access is in progress.

19.1.5 Problem #5: Cast versus Parenthesized Expression

Consider the production:

CastExpression:

`(` *PrimitiveType* `)` *UnaryExpression*

(ReferenceType) UnaryExpressionNotPlusMinus

Now consider the partial input:

```
class Problem5 { Problem5() { super((matthew)
```

When the parser is considering the token `matthew`, with one-token lookahead to symbol `)`, it cannot yet tell whether `(matthew)` will be a parenthesized expression, as in:

```
super((matthew), 9);
```

or a cast, as in:

```
super((matthew)baz, 9);
```

Therefore, after the parser reduces `matthew` to the nonterminal *Name*, it cannot tell with only one-token lookahead whether *Name* should be further reduced to *PostfixExpression* and ultimately to *Expression* (for a parenthesized expression) or to *ClassOrInterfaceType* and then to *ReferenceType* (for a cast). Therefore, the productions shown above result in a grammar that is not LALR(1).

The solution is to eliminate the use of the nonterminal *ReferenceType* in the definition of *CastExpression*, which requires some reworking of both alternatives to avoid other ambiguities:

CastExpression:

(PrimitiveType Dims$_{opt}$ *) UnaryExpression*

(Expression) UnaryExpressionNotPlusMinus

(Name Dims) UnaryExpressionNotPlusMinus

This allows the parser to reduce `matthew` to *Expression* and then leave it there, delaying the decision as to whether a parenthesized expression or a cast is in progress. Inappropriate variants such as:

```
(int[])+3
```

and:

```
(matthew+1)baz
```

must then be weeded out and rejected by a later stage of compiler analysis.

The remaining sections of this chapter constitute a LALR(1) grammar for Java syntax, in which the five problems described above have been solved.

19.2 Productions from §2.3: The Syntactic Grammar

Goal:

CompilationUnit

19.3 Productions from §3: Lexical Structure

Literal:

IntegerLiteral

FloatingPointLiteral

```
        BooleanLiteral

        CharacterLiteral

        StringLiteral

        NullLiteral
```

19.4 Productions from §4: Types, Values, and Variables

```
Type:

        PrimitiveType

        ReferenceType

PrimitiveType:

        NumericType

        boolean

NumericType:

        IntegralType

        FloatingPointType

IntegralType: one of

        byte short int long char

FloatingPointType: one of

        float double

ReferenceType:

        ClassOrInterfaceType

        ArrayType

ClassOrInterfaceType:

        Name

ClassType:

        ClassOrInterfaceType

InterfaceType:

        ClassOrInterfaceType

ArrayType:

        PrimitiveType [ ]

        Name [ ]

        ArrayType [ ]
```

19.5 Productions from §6: Names

Name:

SimpleName

QualifiedName

SimpleName:

Identifier

QualifiedName:

Name . *Identifier*

19.6 Productions from §7: Packages

CompilationUnit:

PackageDeclaration$_{opt}$ *ImportDeclarations*$_{opt}$ *TypeDeclarations*$_{opt}$

ImportDeclarations:

ImportDeclaration

ImportDeclarations ImportDeclaration

TypeDeclarations:

TypeDeclaration

TypeDeclarations TypeDeclaration

PackageDeclaration:

package *Name* ;

ImportDeclaration:

SingleTypeImportDeclaration

TypeImportOnDemandDeclaration

SingleTypeImportDeclaration:

import *Name* ;

TypeImportOnDemandDeclaration:

import *Name* . * ;

TypeDeclaration:

ClassDeclaration

InterfaceDeclaration

;

19.7 Productions Used Only in the LALR(1) Grammar

Modifiers:

Modifier

Modifiers Modifier

Modifier: one of

public protected private

static

abstract final native synchronized transient volatile

19.8 Productions from §8: Classes

19.8.1 Productions from §8.1: Class Declaration

ClassDeclaration:

Modifiers$_{opt}$ class *Identifier Super*$_{opt}$ *Interfaces*$_{opt}$ *ClassBody*

Super:

extends *ClassType*

Interfaces:

implements *InterfaceTypeList*

InterfaceTypeList:

InterfaceType

InterfaceTypeList , *InterfaceType*

ClassBody:

{ *ClassBodyDeclarations*$_{opt}$ }

ClassBodyDeclarations:

ClassBodyDeclaration

ClassBodyDeclarations ClassBodyDeclaration

ClassBodyDeclaration:

ClassMemberDeclaration

StaticInitializer

ConstructorDeclaration

ClassMemberDeclaration:

FieldDeclaration

MethodDeclaration

19.8.2 Productions from §8.3: Field Declarations

FieldDeclaration:

Modifiers$_{opt}$ *Type VariableDeclarators* ;

VariableDeclarators:

VariableDeclarator

VariableDeclarators , *VariableDeclarator*

VariableDeclarator:

VariableDeclaratorId

VariableDeclaratorId = *VariableInitializer*

VariableDeclaratorId:

Identifier

VariableDeclaratorId [ ]

VariableInitializer:

Expression

ArrayInitializer

19.8.3 Productions from §8.4: Method Declarations

MethodDeclaration:

MethodHeader MethodBody

MethodHeader:

Modifiers$_{opt}$ *Type MethodDeclarator Throws*$_{opt}$

Modifiers$_{opt}$ void *MethodDeclarator Throws*$_{opt}$

MethodDeclarator:

Identifier (*FormalParameterList*$_{opt}$)

MethodDeclarator [ ]

FormalParameterList:

FormalParameter

FormalParameterList , *FormalParameter*

FormalParameter:

Type VariableDeclaratorId

Throws:

throws *ClassTypeList*

ClassTypeList:

ClassType

ClassTypeList , *ClassType*

MethodBody:

Block

;

19.8.4 Productions from §8.5: Static Initializers

StaticInitializer:

static *Block*

19.8.5 Productions from §8.6: Constructor Declarations

ConstructorDeclaration:

Modifiers$_{opt}$ *ConstructorDeclarator* *Throws*opt *ConstructorBody*

ConstructorDeclarator:

SimpleName (*FormalParameterList*$_{opt}$)

ConstructorBody:

{ *ExplicitConstructorInvocation*$_{opt}$ *BlockStatements*$_{opt}$ }

ExplicitConstructorInvocation:

this (*ArgumentList*$_{opt}$) ;

super (*ArgumentList*$_{opt}$) ;

19.9 Productions from §9: Interfaces

19.9.1 Productions from §9.1: Interface Declarations

InterfaceDeclaration:

Modifiers$_{opt}$ interface *Identifier* *ExtendsInterfaces*$_{opt}$ *InterfaceBody*

ExtendsInterfaces:

extends *InterfaceType*

ExtendsInterfaces , *InterfaceType*

InterfaceBody:

{ *InterfaceMemberDeclarations*$_{opt}$ }

InterfaceMemberDeclarations:

InterfaceMemberDeclaration

InterfaceMemberDeclarations *InterfaceMemberDeclaration*

InterfaceMemberDeclaration:

ConstantDeclaration

AbstractMethodDeclaration

ConstantDeclaration:

FieldDeclaration

AbstractMethodDeclaration:

MethodHeader ;

19.10 Productions from §10: Arrays

ArrayInitializer:

{ *VariableInitializers*$_{opt}$,$_{opt}$ }

VariableInitializers:

VariableInitializer

VariableInitializers , *VariableInitializer*

19.11 Productions from §14: Blocks and Statements

Block:

{ *BlockStatements*$_{opt}$ }

BlockStatements:

BlockStatement

BlockStatements BlockStatement

BlockStatement:

LocalVariableDeclarationStatement

Statement

LocalVariableDeclarationStatement:

LocalVariableDeclaration ;

LocalVariableDeclaration:

Type VariableDeclarators

Statement:

StatementWithoutTrailingSubstatement

LabeledStatement

IfThenStatement

IfThenElseStatement

WhileStatement

```
        ForStatement

StatementNoShortIf:

        StatementWithoutTrailingSubstatement

        LabeledStatementNoShortIf

        IfThenElseStatementNoShortIf

        WhileStatementNoShortIf

        ForStatementNoShortIf

StatementWithoutTrailingSubstatement:

        Block

        EmptyStatement

        ExpressionStatement

        SwitchStatement

        DoStatement

        BreakStatement

        ContinueStatement

        ReturnStatement

        SynchronizedStatement

        ThrowStatement

        TryStatement

EmptyStatement:

        ;

LabeledStatement:

        Identifier : Statement

LabeledStatementNoShortIf:

        Identifier : StatementNoShortIf

ExpressionStatement:

        StatementExpression ;

StatementExpression:

        Assignment

        PreIncrementExpression

        PreDecrementExpression

        PostIncrementExpression
```

PostDecrementExpression

MethodInvocation

ClassInstanceCreationExpression

IfThenStatement:

if (*Expression*) *Statement*

IfThenElseStatement:

if (*Expression*) *StatementNoShortIf* else *Statement*

IfThenElseStatementNoShortIf:

if (*Expression*) *StatementNoShortIf* else *StatementNoShortIf*

SwitchStatement:

switch (*Expression*) *SwitchBlock*

SwitchBlock:

{ *SwitchBlockStatementGroups*$_{opt}$ *SwitchLabels*$_{opt}$ }

SwitchBlockStatementGroups:

SwitchBlockStatementGroup

SwitchBlockStatementGroups SwitchBlockStatementGroup

SwitchBlockStatementGroup:

SwitchLabels BlockStatements

SwitchLabels:

SwitchLabel

SwitchLabels SwitchLabel

SwitchLabel:

case *ConstantExpression* :

default :

WhileStatement:

while (*Expression*) *Statement*

WhileStatementNoShortIf:

while (*Expression*) *StatementNoShortIf*

DoStatement:

do *Statement* while (*Expression*) ;

ForStatement:

for (*ForInit*$_{opt}$; *Expression*$_{opt}$; *ForUpdate*$_{opt}$)

Statement

ForStatementNoShortIf:

for (*ForInit*$_{opt}$; *Expression*$_{opt}$; *ForUpdate*$_{opt}$)

StatementNoShortIf

ForInit:

StatementExpressionList

LocalVariableDeclaration

ForUpdate:

StatementExpressionList

StatementExpressionList:

StatementExpression

StatementExpressionList , *StatementExpression*

BreakStatement:

break *Identifier*$_{opt}$;

ContinueStatement:

continue *Identifier*$_{opt}$;

ReturnStatement:

return *Expression*$_{opt}$;

ThrowStatement:

throw *Expression* ;

SynchronizedStatement:

synchronized (*Expression*) *Block*

TryStatement:

try *Block* *Catches*

try *Block* *Catches*$_{opt}$ *Finally*

Catches:

CatchClause

Catches *CatchClause*

CatchClause:

catch (*FormalParameter*) *Block*

Finally:

finally *Block*

19.12 Productions from §15: Expressions

Primary:

PrimaryNoNewArray

ArrayCreationExpression

PrimaryNoNewArray:

Literal

this

(*Expression*)

ClassInstanceCreationExpression

FieldAccess

MethodInvocation

ArrayAccess

ClassInstanceCreationExpression:

new *ClassType* (*ArgumentList*$_{opt}$)

ArgumentList:

Expression

ArgumentList , *Expression*

ArrayCreationExpression:

new *PrimitiveType DimExprs Dims*$_{opt}$

new *ClassOrInterfaceType DimExprs Dims*$_{opt}$

DimExprs:

DimExpr

DimExprs DimExpr

DimExpr:

[*Expression*]

Dims:

[ ]

Dims [ ]

FieldAccess:

Primary . *Identifier*

super . *Identifier*

MethodInvocation:

Name (*ArgumentList*$_{opt}$)

Primary . *Identifier* (*ArgumentList*$_{opt}$)

super . *Identifier* (*ArgumentList*$_{opt}$)

ArrayAccess:

Name [*Expression*]

PrimaryNoNewArray [*Expression*]

PostfixExpression:

Primary

Name

PostIncrementExpression

PostDecrementExpression

PostIncrementExpression:

PostfixExpression ++

PostDecrementExpression:

PostfixExpression --

UnaryExpression:

PreIncrementExpression

PreDecrementExpression

+ *UnaryExpression*

- *UnaryExpression*

UnaryExpressionNotPlusMinus

PreIncrementExpression:

++ *UnaryExpression*

PreDecrementExpression:

-- *UnaryExpression*

UnaryExpressionNotPlusMinus:

PostfixExpression

~ *UnaryExpression*

! *UnaryExpression*

CastExpression

CastExpression:

(*PrimitiveType Dims*$_{opt}$) *UnaryExpression*

```
        ( Expression ) UnaryExpressionNotPlusMinus

        ( Name Dims ) UnaryExpressionNotPlusMinus

MultiplicativeExpression:

        UnaryExpression

        MultiplicativeExpression * UnaryExpression

        MultiplicativeExpression / UnaryExpression

        MultiplicativeExpression % UnaryExpression

AdditiveExpression:

        MultiplicativeExpression

        AdditiveExpression + MultiplicativeExpression

        AdditiveExpression - MultiplicativeExpression

ShiftExpression:

        AdditiveExpression

        ShiftExpression << AdditiveExpression

        ShiftExpression >> AdditiveExpression

        ShiftExpression >>> AdditiveExpression

RelationalExpression:

        ShiftExpression

        RelationalExpression < ShiftExpression

        RelationalExpression > ShiftExpression

        RelationalExpression <= ShiftExpression

        RelationalExpression >= ShiftExpression

        RelationalExpression instanceof ReferenceType

EqualityExpression:

        RelationalExpression

        EqualityExpression == RelationalExpression

        EqualityExpression != RelationalExpression

AndExpression:

        EqualityExpression

        AndExpression & EqualityExpression

ExclusiveOrExpression:

        AndExpression
        ExclusiveOrExpression ^ AndExpression

InclusiveOrExpression:

        ExclusiveOrExpression

        InclusiveOrExpression | ExclusiveOrExpression

ConditionalAndExpression:

        InclusiveOrExpression

        ConditionalAndExpression && InclusiveOrExpression

ConditionalOrExpression:

        ConditionalAndExpression

        ConditionalOrExpression || ConditionalAndExpression

ConditionalExpression:

        ConditionalOrExpression

        ConditionalOrExpression ? Expression : ConditionalExpression

AssignmentExpression:

        ConditionalExpression

        Assignment

Assignment:

        LeftHandSide AssignmentOperator AssignmentExpression

LeftHandSide:

        Name

        FieldAccess

        ArrayAccess

AssignmentOperator: one of

        = *= /= %= += -= <<= >>= >>>= &= ^= |=

Expression:

        AssignmentExpression

ConstantExpression:

        Expression
```

Contents | Prev | Next | Index

Java Language Specification (HTML generated by dkramer on August 01, 1996)

Please send any comments or corrections to doug.kramer@sun.com