Appendix 4 The Grammar For Java

It is time to look at a real grammar. Here is a grammar for Java 1.1. It uses a somewhat different syntax, in that it doesn't show the "|", and it uses the *Opt* suffix to represent an optional occurrence of something.

Tokens

Keywords

"abstract" "boolean" "break" "byte" "case" "catch" "char" "class" "continue" "default" "do" "double" "else" "extends" "final" "finally" "float" "for" "if" "implements" "import" "instanceof" "int" "interface" "long" "native" "new" "package" "private" "protected" "public" "return" "short" "static" "super" "switch" "synchronized" "this" "throw" "throws" "transient" "try" "void" "volatile" "while"

Special Symbols

"(" ")" "{" "}" "[" "]" ";" "," "."

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

"<" ">" "<=" ">=" "==" "!=" "!" "~" "?" ":" "&&" "||"
"++" "--" "+" "-" "*" "/" "&" "|" "^"
"%" "<<" ">>" ">>>"

Identifiers

IDENTIFIER

An identifier is a letter, followed by zero or more letters or digits. Letters include "A" .. "Z", "a" .. "z" and the dollar "$" and underscore "_" characters, together with various non ascii characters, such as 猫 or ネコ. Digits are "0" .. "9".

Literals

INTEGERLIT FLOATLIT BOOLEANLIT CHARLIT STRINGLIT NULLLIT

An integer literal can be "0", or a decimal, hexadecimal, or octal literal. A decimal literal is nonzero decimal digit, followed by zero or more decimal digits. A hexadecimal literal is "0x" or "0X" followed by one or more hexadecimal digits. An octal literal is "0" followed by one or more octal digits. Decimal digits are "0" .. "9", hexadecimal digits are "0" .. "9", "A" .. "F", "a" .. "f", octal digits are "0" .. "7". An integer literal can be followed by "l" or "L" to explicitly make it represent a long integer.

A float literal can be one of

Digits "." Digits*Opt* ExponentPart*Opt*

"." Digits ExponentPart*Opt*

Digits ExponentPart

Where ExponentPart is

ExponentIndicator Sign$_{Opt}$ Digits

Digits represents one or more decimal digits, ExponentIndicator is “E” or “e”, Sign is a “+” or “-”. A float literal can be followed by “f”, “F”, “d” or “D”, to explicitly make it a float or double literal.

A Boolean literal is “true” or “false”.

A character literal is one of

“'” Character “'”

“'” EscapeCharacter “'”

where Character is any input character except “\” or “'”, and EscapeCharacter is a “\” followed by a “b”, “t”, “n”, “f”, “"”, “'”, or “\”, or a “\” followed by up to three octal digits.

A string literal is

“"” StringCharacters$_{Opt}$ “"”

where StringCharacters is a sequence of “StringCharacters”s, and a StringCharacter is any input character other than “\” or “"”, or an EscapeCharacter.

A null literal is just “null”.

Grammar Rules

A grammar rule is written in the form

Construct →
Alternative1
Alternative2
Alternative3

where each alternative is a sequence of tokens, literals, and other constructs. Essentially this says that the construct can match any one of the alternatives. If we append $_{Opt}$ to a construct name, as in Modifiers$_{Opt}$, then we mean that the construct is optional.

Example 1

If we take the grammar rules

Type →
PrimitiveType
ReferenceType

PrimitiveType →
NumericType
“boolean”

NumericType →
IntegralType
FloatingPointType

IntegralType →
“byte”

"short"
"int"
"long"
"char"

FloatingPointType →
"float"
"double"

ReferenceType →
ClassOrInterfaceType
ArrayType

ClassOrInterfaceType →
Name

ArrayType →
PrimitiveType "[" "]"
ClassOrInterfaceType "[" "]"
ArrayType "[" "]"

We can see that a Type can be a PrimitiveType, which can be a NumericType, which can be an IntegralType, which can be "int" or "char", etc. Similarly, we can match Type to "double".

A Type can also be a ReferenceType, which can be an ArrayType, which can be PrimitiveType "[" "]", and PrimitiveType can be "int", so a Type can also be "int[]".

Example 2

If we take the grammar rules

Name →
IDENTIFIER
Name "." IDENTIFIER

We can see that a Name can be an IDENTIFIER, for example "System".

A Name can also be Name "." IDENTIFIER, and this Name can be an IDENTIFIER. Hence the overall name can be

IDENTIFIER "." IDENTIFIER

for example, "System.out".

Can you see how we can match Name to "System.out.println", using these grammar rules?

Java Program

CompilationUnit

Literals, Types, Values and Variables

Literals

Literal →
- INTEGERLIT
- FLOATLIT
- BOOLEANLIT
- CHARLIT
- STRINGLIT
- NULLLIT

Types

Type →
- PrimitiveType
- ReferenceType

PrimitiveType →
- NumericType
- “boolean”

NumericType →
- IntegralType
- FloatingPointType

IntegralType →
- “byte”
- “short”
- “int”
- “long”
- “char”

FloatingPointType →
- “float”
- “double”

ReferenceType →
- ClassOrInterfaceType
- ArrayType

ClassOrInterfaceType →
- Name

ClassType →
- ClassOrInterfaceType

InterfaceType →
- ClassOrInterfaceType

ArrayType →

PrimitiveType “[” “]”
ClassOrInterfaceType “[” “]”
ArrayType “[” “]”

Names

Name →
IDENTIFIER
Name “.” IDENTIFIER

Modifiers

Modifiers →
Modifier
Modifiers Modifier

Modifier →
“public”
“protected”
“private”
“static”
“abstract”
“final”
“native”
“symchronized”
“transient”
“volatile”

Packages

CompilationUnit →
PackageDeclaration$_{Opt}$ ImportDeclarations$_{Opt}$ TypeDeclarations$_{Opt}$

ImportDeclarations →
ImportDeclaration
ImportDeclarations ImportDeclaration

TypeDeclarations →
TypeDeclaration
TypeDeclarations TypeDeclaration

PackageDeclaration →
“package” Name “;”

ImportDeclaration →
“import” Name “;”
“import” Name “.” “*” “;”

TypeDeclaration →
ClassDeclaration
InterfaceDeclaration
“;”

Classes

Class Declarations

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 →
    FieldDeclaration
    MethodDeclaration
    ConstructorDeclaration
    ClassDeclaration
    InterfaceDeclaration
    “static” Block
    Block

Field Declarations

FieldDeclaration →
    Modifiers*Opt* Type VariableDeclarators “;”

VariableDeclarators →
    VariableDeclarator
    VariableDeclarators “,” VariableDeclarator

VariableDeclarator →
    VariableDeclaratorId
    VariableDeclaratorId “=” VariableInitializer

VariableDeclaratorId →
    IDENTIFIER
    VariableDeclaratorId “[” “]”

VariableInitializer →
    Expression
    ArrayInitializer

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 →
    Modifiers*Opt* Type VariableDeclaratorId

Throws →
    "throws" ClassTypeList

ClassTypeList →
    ClassType
    ClassTypeList "," ClassType

MethodBody →
    Block
    ";"

Constructor Declarations

ConstructorDeclaration →
    Modifiers*Opt* ConstructorDeclarator Throws*Opt* ConstructorBody

ConstructorDeclarator →
    IDENTIFIER "(" FormalParameterList*Opt* ")"

ConstructorBody →
    "{" ExplicitConstructorInvocation*Opt* BlockStatements*Opt* "}"

ExplicitConstructorInvocation →
    "this" "(" ArgumentList*Opt* ")" ";"
    "super" "(" ArgumentList*Opt* ")" ";"
    Primary "." "super" "(" ArgumentList*Opt* ")" ";"

Interfaces

Interface Declarations

InterfaceDeclaration →
    Modifiers*Opt* “interface” IDENTIFIER ExtendsInterfaces*Opt* InterfaceBody

ExtendsInterfaces →
    “extends” InterfaceTypeList

InterfaceBody →
    “{” InterfaceMemberDeclarations*Opt* “}”

InterfaceMemberDeclarations →
    InterfaceMemberDeclaration
    InterfaceMemberDeclarations InterfaceMemberDeclaration

InterfaceMemberDeclaration →
    ConstantDeclaration
    AbstractMethodDeclaration
    ClassDeclaration
    InterfaceDeclaration

ConstantDeclaration →
    FieldDeclaration

AbstractMethodDeclaration →
    MethodHeader “;”

Arrays

ArrayInitializer →
    “{” “}”
    “{” VariableInitializers “}”
    “{” VariableInitializers “,” “}”

VariableInitializers →
    VariableInitializer
    VariableInitializers “,” VariableInitializer

Blocks and Statements

Block →
    “{” BlockStatements*Opt* “}”

BlockStatements →
    BlockStatement
    BlockStatements BlockStatement

BlockStatement →

LocalVariableDeclaration ";"
ClassDeclaration
Statement

LocalVariableDeclaration →
ModifiersOpt 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$_{Opt}$ “;” ExpressionOpt “;” ForUpdateOpt “)” Statement

ForStatementNoShortIf →

“for” “(” ForInitOpt “;” ExpressionOpt “;” ForUpdateOpt “)”
StatementNoShortIf

ForInit→
StatementExpressionList
LocalVariableDeclaration

ForUpdate→
StatementExpressionList

StatementExpressionList→
StatementExpression
StatementExpressionList “,” StatementExpression

BreakStatement→
“break” IDENTIFIER “;”
“break” “;”

ContinueStatement→
“continue” IDENTIFIER “;”
“continue” “;”

ReturnStatement→
“return” Expression ;”
“return” “;”

ThrowStatement→
“throw” Expression “;”

SynchronizedStatement→
“synchronized” “(” Expression “)” Block

TryStatement→
“try” Block Catches
“try” Block Finally
“try” Block Catches Finally

Catches→
CatchClause
Catches CatchClause

CatchClause→
“catch” “(” FormalParameter “)” Block

Finally→
“finally” Block

Expressions

Primary→

PrimaryNoNewArray
“new” PrimitiveType DimExprs DimsOpt
“new” ClassOrInterfaceType DimExprs DimsOpt
“new” Type Dims ArrayInitializer

PrimaryNoNewArray→
Literal
“this”
ClassName “.” “this”
“(” Expression “)”
ClassInstanceCreationExpression
FieldAccess
MethodInvocation
ArrayAccess
Type “.” “class”
“void” “.” “class”

ClassInstanceCreationExpression→
“new” TypeName “(” ArgumentListOpt “)” ClassBodyOpt
Primary “.” “new” Identifier “(” ArgumentListOpt “)” ClassBodyOpt

ArgumentList→
Expression
ArgumentList “,” Expression

DimExprs→
DimExpr
DimExprs DimExpr

DimExpr→
“[” Expression “]”

Dims→
“[” “]”
Dims “[” “]”

FieldAccess→
Primary “.” IDENTIFIER
“super” “.” IDENTIFIER

MethodInvocation→
Name “(” ArgumentListOpt “)”
Primary “.” IDENTIFIER “(” ArgumentListOpt “)”
“super” “.” IDENTIFIER “(” ArgumentListOpt “)”

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

RelationExpression →
ShiftExpression
RelationExpression “<” ShiftExpression
RelationExpression “>” ShiftExpression
RelationExpression “<=” ShiftExpression
RelationExpression “>=” ShiftExpression
RelationExpression “instanceof” ReferenceType

EqualityExpression →
RelationExpression
EqualityExpression “==” RelationExpression
EqualityExpression “!=” RelationExpression

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 "="AssignmentExpression
    LeftHandSide "*="AssignmentExpression
    LeftHandSide "/="AssignmentExpression
    LeftHandSide "%="AssignmentExpression
    LeftHandSide "+="AssignmentExpression
    LeftHandSide "-="AssignmentExpression
    LeftHandSide "<<="AssignmentExpression
    LeftHandSide ">>="AssignmentExpression
    LeftHandSide ">>>="AssignmentExpression
    LeftHandSide "&="AssignmentExpression
    LeftHandSide "|="AssignmentExpression

LeftHandSide →
    Name
    FieldAccess
    ArrayAccess

Expression →
    AssignmentExpression

ConstantExpression →
    Expression