

Jim Lambers
CS143 Compilers
Summer Quarter 2000-01
Mocha Language Specification

1 Overview

This document contains the complete specification for the Mocha language, for which you will write a compiler during this course. Mocha is essentially a subset of Java, with some slight modifications in order to simplify parser generation. The semantics are very similar to that of Java, where applicable.

2 Lexical Structure

In this section, we define all of the tokens of Mocha, and specify what constitutes a valid lexeme for each token. In addition, we indicate what attributes should be supplied with each token, if any. These attributes provide additional information about the token that is necessary for further stages of compilation.

2.1 White Space

White space, often ignored between other tokens, is defined to be the ASCII space, horizontal tab, and form feed characters, as well as the *line terminators*, which are the ASCII linefeed character, the ASCII carriage return character, or the two-character sequence of a carriage-return followed by a linefeed. Under no circumstances is whitespace returned as a token.

2.2 Comments

A *comment* is a sequence of characters that is intended to be ignored by the compiler, thereby allowing one to add arbitrary text to a Mocha program, often for descriptive purposes. Mocha allows two types of comments: a single-line comment and a multi-line comment.

A single-line comment consists of two forward slashes (//) and all text following the slashes, up to and including the next line terminator.

A multi-line comment consists of the starting delimiter /* and all text until the ending delimiter */ is encountered, including line terminators. In Mocha, a starting delimiter may occur within a comment, but it will be

ignored. As a result, multi-line comments may not be nested within one another.

Comments are to be ignored during syntax analysis, and therefore no token is to be returned upon recognizing a comment.

2.3 Identifiers

An identifier is a sequence of Mocha letters and Mocha digits, with the restriction that the first character in the sequence must be a Mocha letter. A Mocha letter is either an uppercase (A-Z) or lowercase (a-z) ASCII Latin letter, an underscore (_), or dollar sign (\$). A Mocha digit is any decimal digit (0-9).

Whenever recognizing an identifier, the lexical analyzer is to return the token `T_identifier`. In addition, the text of the identifier name is to be returned.

2.4 Keywords

The following words are reserved words, or *keywords*. As such, they may not be used as identifiers.

boolean	break	byte
case	char	class
continue	default	do
double	else	extends
final	float	for
if	instanceof	int
long	new	private
protected	public	return
short	static	switch
this	void	while

Each keyword is its own token, and its token name is obtained by appending the keyword itself to the `T_` prefix. Therefore, the token for `if` is `T_if`, for example.

Keywords do not have attributes.

2.5 Literals

2.5.1 Integer Literals

An integer literal consists of a *numeral*, which may be expressed in decimal, octal, or hexadecimal form.

A *decimal numeral* is either the single digit 0, or a sequence of decimal digits (0–9), the first of which is non-zero.

An *octal numeral* is a 0, followed by a sequence of one or more octal digits (0–7).

A *hexadecimal numeral* is the digit 0, followed by the letter X or x, and a sequence of one or more hexadecimal digits. Hexadecimal digits include the decimal digits, and the letters a–f or A–F, to represent the values 10 through 15.

All integer literals are lexemes for the token `T_intliteral`. The attribute for this token is the literal’s value, represented as a `long`.

2.5.2 Floating-point Literals

A floating-point literal consists of a sequence of decimal digits that may include a decimal point, and an optional exponent part.

The sequence of digits with optional decimal point must conform to the following rules:

- There can be at most one decimal point. A decimal point is not required, however.
- There must be at least one decimal digit, which may occur on either side of the decimal point.

The exponent part consists of an exponent indicator, which is the letter E or e, followed by an optional sign (plus or minus), and an unsigned integer specified using one or more decimal digits. If the sign is absent, the exponent is assumed to be positive.

All floating-point constants are lexemes for the token `T_floatliteral`. The attribute for this token is the literal’s value, represented as a double-precision floating-point number, stored in a variable of type `double`.

2.5.3 Boolean Literals

A boolean literal has two values, represented by the words `true` and `false`. Like keywords, these words may not be used as identifiers. Both values are lexemes for the token `T_boolliteral`, which has as its attribute a `bool` value equal to 1 for the literal `true` and 0 for the literal `false`.

2.5.4 Character Literals

A character literal consists of a single character, or an escape sequence, enclosed in single quotes (`'`). The single character may not be a line terminator, a single quote, or a backslash (`\`). Escaped sequences are discussed below. The token for character literals is `T_charliteral`. Its attribute is a `char` whose value is the character represented by the literal.

2.5.5 String Literals

A string literal consists of a sequence of characters or escape sequences enclosed in double quotes (`"`). Line terminators, double quotes, and backslashes cannot be included in a string literal except by using the appropriate escape sequences, which are discussed below. The token for string literals is `T_stringliteral`. The attribute for this token is the string literal itself, stored in a `char` array.

2.5.6 Escape Sequences

An *escape sequence* may be used within a character or string literal. An escape sequence consists of a backslash (`\`), followed by either an octal value from 0 to 0377, or any of the characters listed in the table below. The characters represented by the escape sequences are indicated.

Character	ASCII Code (hex)	Meaning
<code>b</code>	0008	backspace
<code>t</code>	0009	horizontal tab
<code>n</code>	000a	linefeed
<code>f</code>	000c	formfeed
<code>r</code>	000d	carriage return
<code>"</code>	0022	double quote
<code>'</code>	0027	single quote
<code>\</code>	005c	backslash

2.5.7 The Null Literal

The null type has a single value, represented by the word `null`. This word may not be used as an identifier. The token for this literal is `T_nullliteral`. It has no attribute.

2.6 Separators

The following characters are the *separators*, also known as *punctuators*:

Character	Token name
(T_lpar
)	T_rpar
{	T_lbrace
}	T_rbrace
[T_lsquare
]	T_rsquare
;	T_semi
,	T_comma
.	T_dot

These tokens have no attributes.

2.7 Operators

The following character sequences are the **operators**. The categories of operators are described in order from highest to lowest precedence. All operators are left associative, except for equality operators, relational operators, and assignment operators, which are not associative, and the ternary operator, which is right associative.

The table below associates various lexemes with attribute values. When a lexeme for an operator is recognized, the corresponding attribute value, if one is specified, must be returned by the lexical analyzer. Tokens that have only one lexeme, such as **T_and**, will not have an attribute. The attribute values themselves are constants. They are defined and assigned numeric values by the implementation of the lexical analyzer.

Character Sequence	Token name	Token Attribute
Prefix Operators		
++	T_incr	A_add
--	T_incr	A_sub
!	T_not	A_not
~	T_not	A_bit
Multiplicative Operators		
*	T_mulop	A_mul
/	T_mulop	A_div
%	T_mulop	A_mod
Additive Operators		
+	T_addop	A_add
-	T_addop	A_sub
Shift Operators		
<<	T_shifto	A_lsh
>>	T_shifto	A_rsh
Relational Operators		
<	T_relop	A_lt
>	T_relop	A_gt
<=	T_relop	A_leq
>=	T_relop	A_geq
Equality Operators		
==	T_eqop	A_eq
!=	T_eqop	A_neq
Bitwise AND Operator		
&	T_bitand	
Bitwise OR Operators		
	T_bitor	A_or
^	T_bitor	A_xor
AND Operator		
&&	T_and	
OR Operator		
	T_or	
Ternary Operator Characters		
:	T_colon	
?	T_quest	

Character Sequence	Token name	Token Attribute
Assignment Operators		
=	T_assign	A_eq
+=	T_assign	A_add
-=	T_assign	A_sub
*=	T_assign	A_mul
/=	T_assign	A_div
%=	T_assign	A_mod
&=	T_assign	A_and
=	T_assign	A_or
^=	T_assign	A_xor
<<=	T_assign	A_lsh
>>=	T_assign	A_rsh

2.8 The Scope Resolution Operator

The *scope resolution operator* is borrowed from the syntax of C++, and is used to separate identifiers in qualified class names. Its lexeme is two colons, ::, and it has no attribute.

3 Syntax

In this section, we specify the complete syntax of Mocha. The syntactic rules are written using conventions similar to those of EBNF (Extended Backus-Naur Form):

- { x } means “zero or more occurrences of x ”
- [x] means “zero or one occurrences of x ”

The terminal symbols of the grammar described below are the tokens defined in the previous section.

3.1 Top-level Constructs

A valid Mocha source program consists of a single *compilation unit*, which itself consists zero or more *class declarations*.

CompilationUnit :=
{ *ClassDeclaration* }

3.2 Classes

Classes are declared using the `class` keyword. The class name must be an identifier. A class may *inherit* from one or more other *base types*; this inheritance is indicated using the `extends` keyword, followed by a comma-separated list of class names.

Classes are referenced using a syntax similar to that of C++. The `class` keyword must precede the class name, which consists of one or more identifiers separated by two colons, the scope resolution operator.

The definition of a class is concluded with the class body, which is a block enclosed in curly braces and containing class body declarations.

ClassDeclaration :=

`T_class T_identifier [Extends] ClassBody`

Extends :=

`T_extends TypeList`

TypeList :=

`Type { T_comma Type }`

Type :=

`BasicType`

| `T_class ClassIdentifier`

BasicType :=

`T_byte`

| `T_short`

| `T_char`

| `T_int`

| `T_long`

| `T_float`

| `T_double`

| `T_boolean`

| `T_void`

ClassIdentifier :=

`T_identifier { T_sro T_identifier }`

ClassBody :=

$T_lbrace \{ \textit{ClassBodyDeclaration} \} T_rbrace$

3.3 Class Members

Classes contain *members*, which may be methods or variables, just as in C++ or Java. They may also contain a *constructor*, which, syntactically, is nearly identical to a method but must not have a return type.

All members may be declared using one or more *modifiers* that control access to the member or its value. These modifiers have the same meaning as their counterparts in Java.

$\textit{ClassBodyDeclaration} :=$
 T_semi
 $| \{ \textit{Modifier} \} \textit{MemberDecl}$

$\textit{Modifier} :=$
 T_public
 $| T_protected$
 $| T_private$
 $| T_static$
 $| T_final$

$\textit{MemberDecl} :=$
 $\textit{MethodDecl}$
 $| \textit{Type} \textit{VariableDeclarator} T_semi$
 $| T_identifier \textit{FormalParameters} \textit{Block}$

3.3.1 Member Variables

Member variables, including those which are arrays, may be declared with optional initializers.

$\textit{VariableDeclarator} :=$
 $T_identifier \{ \textit{BracketPair} \} [\textit{VariableInitializer}]$

$\textit{BracketPair} :=$
 $T_lsquare T_rsquare$

$\textit{VariableInitializer} :=$

Expression
 | *ArrayInitializer*

ArrayInitializer :=
T_lbrace [*VariableInitializers*] **T_rbrace**

VariableInitializers :=
VariableInitializer { **T_comma** *VariableInitializer* }

3.3.2 Methods

Class methods have the same syntax as in Java, consisting of a return type, an identifier representing the method name, a list of zero or more formal parameters, and a block containing statements.

MethodDecl :=
Type **T_identifier** *FormalParameters* *Block*

FormalParameters :=
T_lpar [*FormalParameterList*] **T_rpar**

FormalParameterList :=
FormalParameter { **T_comma** *FormalParameter* }

FormalParameter :=
 [*T_final*] *Type* *VariableDeclaratorId*

VariableDeclaratorId :=
T_identifier { *BracketPair* }

3.4 Blocks and Statements

The body of a class method is a single statement block. We now define the syntactic structure of a block and the various statements it may contain.

3.4.1 Blocks

A statement block is delimited by curly braces, and contains a list of zero or more statements.

Block :=
 T_lbrace { *BlockStatement* } T_rbrace

BlockStatement :=
 LocalVarDeclStmt
 | *Statement*

3.4.2 Variable Declaration Statements

A block may include statements that are used to declare variables that are local to the block. Like the member variables of a class, local variables may also be declared with initializers.

LocalVarDeclStmt :=
 [T_final] *Type* *VariableDeclarators* T_semi

VariableDeclarators :=
 VariableDeclarator { T_comma *VariableDeclarator* }

3.4.3 Statements

Mocha offers the most familiar statements from C++ and Java for evaluation of expressions and flow control.

Statement :=
 Block
 | T_if *ParExpression* *Statement* T_else *Statement*
 | T_if *ParExpression* *Statement*
 | T_for T_lpar [*ExpressionList*] T_semi
 [*Expression*] T_semi [*ExpressionList*] T_rpar *Statement*
 | T_while *ParExpression* *Statement*
 | T_do *Statement* T_while *ParExpression* T_semi
 | T_switch *ParExpression* T_lbrace { *SwitchBlockStmtGroup* } T_rbrace
 | T_return T_semi
 | T_return *Expression* T_semi
 | T_break T_semi
 | T_continue T_semi
 | T_semi
 | *Expression* T_semi

ParExpression :=
 T_lpar *Expression* T_rpar

3.4.4 Switch Blocks

A **switch** statement may contain zero or more blocks, each of which must begin with a **case** label or a **default** label. The block itself contains zero or more statements.

SwitchBlockStmtGroup :=
 SwitchLabel { *BlockStatement* }

SwitchLabel :=
 T_case *Expression* T_colon
 | T_default T_colon

3.5 Expressions

Expressions are the lowest-level constructs in Mocha. We describe the various types of expressions starting with the operators with lowest precedence.

3.5.1 High-level Expressions

The top-level expressions consist of assignments and the use of the ternary operator.

Expression :=
 Expression1 [T_assign *Expression1*]

Expression1 :=
 Expression2 [T_quest *Expression1* T_colon *Expression1*]

3.5.2 Infix Operators

The infix operators have precedence defined in the section of lexical rules. In addition, the keyword **instanceof**, represented by the terminal symbol **T_instanceof**, is an infix operator, with higher precedence than any other

infix operators. These precedence rules help resolve syntactic ambiguities. All infix operators are left associative, except for equality operators, relational operators, and the `instanceof` operator, which are not associative.

Expression2 :=
 Expression2 `T_mulop` *Expression2*
 | *Expression2* `T_addop` *Expression2*
 | *Expression2* `T_relop` *Expression2*
 | *Expression2* `T_shifto` *Expression2*
 | *Expression2* `T_eqop` *Expression2*
 | *Expression2* `T_and` *Expression2*
 | *Expression2* `T_or` *Expression2*
 | *Expression2* `T_bitand` *Expression2*
 | *Expression2* `T_bitor` *Expression2*
 | *Expression2* `T_instanceof` *Expression2*
 | *Expression3*

3.5.3 Prefix Operators

The prefix operators include pre-increment and pre-decrement, negation, addition and subtraction, and casting to a type.

Expression3 :=
 PrefixOp *Expression3*
 | *Expression4*

PrefixOp :=
 `T_incr`
 | `T_not`
 | `T_addop`
 | `T_lpar` *Type* `T_rpar`

3.5.4 Postfix Operators

The postfix operators include post-increment and post-decrement, method invocation, member selection, and array element access.

Expression4 :=
 Expression4 *PostfixOp*

| *Primary*

PostfixOp :=

 T_incr
 | *Arguments*
 | *Selector*

Arguments :=

 T_lpar [*ExpressionList*] T_rpar

ExpressionList :=

Expression { T_comma *Expression* }

Selector :=

 T_dot T_identifier
 | T_lsquare *Expression* T_rsquare

3.5.5 Primary Expressions

The primary expressions are the lowest-level expressions, consisting of variables (which may be static members of another class), literals, parenthesized expressions, and object construction using **new**.

Primary :=

 T_lpar *Expression* T_rpar
 | T_this
 | *Literal*
 | *QualifiedIdentifier*
 | T_new *Type*

Literal :=

 T_intliteral
 | T_floatliteral
 | T_charliteral
 | T_stringliteral
 | T_boolliteral
 | T_nullliteral

QualifiedIdentifier :=

```

T_identifier
| T_sro ClassIdentifier T_dot T_identifier

```

4 Semantics

This section describes a syntax-directed definition for performing semantic analysis on a Mocha program. This definition assigns attributes to many of the various non-terminals, which in turn are used for semantic checks such as type checking.

4.1 The Mocha Type System

Here, we describe the types supported by Mocha, including the basic, atomic types and the more complex types that can be built using supplied *constructors*. We also specify the rules used to determine whether two type expressions are compatible.

4.1.1 Basic Types

Mocha supports the following basic types:

```

BasicType :=
  T_byte
  | T_short
  | T_char
  | T_int
  | T_long
  | T_float
  | T_double
  | T_boolean
  | T_void

```

In addition, the language includes the `null` type, with the single value `null`, and a type `none`, for internal use only. This is the type assigned to the constructor of a class.

4.1.2 Type Constructors

The following type constructors are used to build type expressions from the ground up, using the basic types from the previous section as the lowest-level building blocks.

- The *array* constructor takes as arguments a *base* type, which may be any type other than `null`, *list*, or *method*, and a number *n*. This constructor returns a new type that is an *n*-dimensional array, each element of which is of the *base* type.
- The *list* constructor takes as arguments a sequence of types, and assembles them into an ordered list. Such a type is used by methods to verify whether they are being invoked properly.
- The *method* constructor takes two arguments, a *return* type and an *argument* type, which must be a *list*. This constructor returns a new type, a *method* type, which is associated with a method that accepts arguments compatible with the *argument* type and returns an object of the *return* type.
- The *class* constructor takes three arguments: a *name* and an optional *base* type. The result of this constructor is a *reference* or *class* type, shared by all objects of the class *name*, which derives from the *base* type. The *base* type must also be a *reference* type.

4.1.3 Classification of Types

For convenience, we use the following terms to denote groups of types.

- The *integral* types are `byte`, `char`, `short`, `int`, and `long`.
- The *numeric* types are the *integral* types, as well as `float` and `double`.
- Any class type is called a *reference* type, and an expression that is of any class type is said to be a *reference*.

4.1.4 Type Compatibility

We say that a type t_1 is *compatible* with another type t_2 if an expression of type t_1 can be substituted wherever an expression of type t_2 is expected. The following rules determine whether t_1 is compatible with t_2 . Note that for our purposes, compatibility is a one-way street. In most cases, it does in fact turn out to be a two-way street, but not all. For example, if t_1 represents a class, and t_2 represents a derived class, then by the rules below, t_2 is compatible with t_1 , but not the other way around.

- If t_2 is a basic type or a *reference* type, then the following rules apply, in order:

- If t_1 is a *method* type, then t_1 is compatible if and only if the return type of the method is compatible.
 - If t_1 is an *array*, then it is not compatible unless t_2 is `null`.
 - If t_1 is `null`, then it is compatible if and only if t_2 is `null` or is a *reference* type.
 - If t_1 is a *reference* type, then t_2 must either be `null`, or it must also be a *reference* type, in which case t_1 's class must derive from that of t_2 , or be the same class.
 - Otherwise, both types must be the same basic type.
- If t_2 is an *array* type, the following rules apply:
 - If t_1 is `null`, then it is compatible.
 - If t_1 is not an array, then it is not compatible.
 - If t_1 is an array, then it is compatible if and only if the *base* type of t_1 is compatible with that of t_2 , and the number of array dimensions of t_1 and t_2 are equal.
 - If t_2 is a *list* type, then t_1 must also be a list type, and for each element x in t_2 's list, the corresponding element y in t_1 's list must be compatible with x .
 - If t_2 is a *method* type, then the following rules apply:
 - If t_1 is a *method* type, then its return type must be compatible with that of t_2 .
 - Otherwise, t_1 must be compatible with the return type of t_2 .

4.1.5 Coercion for Numeric Types

For various binary operations involving the numeric types, the result type is determined by coercing one of the operands to change type, to that of the operand. To determine the proper coercion, we *rank* all of the numeric types from lowest to highest. Then, a coercion involves converting the operand with the lower-ranked type to an operand with the higher-ranked type.

The numeric types are ranked in the following order, from lowest to highest: `byte`, `char`, `short`, `int`, `long`, `float`, `double`.

4.2 Scoping

Mocha programs contain three scopes: *Global*, *Class*, and *Method*. The scope defines a realm within which a name is known. All class names in a Mocha source file reside within the *Global* scope, and no other names are defined within that scope. All names of members of a class have *Class* scope; these names are only recognized through a reference to that class. For each method, the names of the formal parameters of the method and all local variables declared within the method have *Method* scope, and are not known outside of that method.

Note that because all local variables and formal parameters of a method have names contained within the same scope, they may only be defined once within that method. Therefore, local variables contained within different blocks in the same method may not share the same name.

Wherever an identifier appears in a method as a stand-alone variable (as opposed to a component in a class identifier, or a member of a class instance), the compiler must first look up the name within the currently active *Method* scope. If it is not found there, it must look in the currently active *Class* scope. If it still is not found, it must look to the scope of the current class' base class, and its base class, and so on, until there is no base class to examine. The *Global* scope is only examined when looking specifically for a class name.

4.3 Declarations

The attributes for non-terminals used to describe declarations are computed as follows:

$CompilationUnit := \{ \textit{ClassDeclaration} \}$

CompilationUnit has a single synthesized attribute, which is a list of class declarations.

$ClassDeclaration := \textit{T_class T_identifier [Extends] ClassBody}$

A *ClassDeclaration* has as a synthesized attribute the declaration of a class, which includes the given class name, an optional base class supplied by *Extends*, and the attribute of *ClassBody*, which is a list of member declarations.

$Extends := \textit{T_extends Type}$

The type denoted by *Type* must be a class, not a basic type. If a basic

type is given, or if the type is a class that has not been defined, an error is to be reported and the synthesized attribute of *Extends* is NULL. Otherwise, this attribute is the class' declaration.

Type := *BasicType*

The synthesized attribute of *Type* is the attribute of *BasicType*, which is an object describing the basic type.

Type := T_class *ClassIdentifier*

The synthesized attribute of *Type* is a type object denoting a class declaration to the class named by *ClassIdentifier*. If such a class has not been defined, an error must be reported, and the attribute *Type* is the type *int*.

ClassIdentifier := T_identifier { T_sro T_identifier }

The synthesized attribute of *ClassIdentifier* is a string consisting of either a single identifier or a sequence of identifiers separated by the scope resolution operator.

ClassBody := T_lbrace { *ClassBodyDeclaration* } T_rbrace

The synthesized attribute of *ClassBody* is a list of the attributes of each *ClassBodyDeclaration*.

ClassBodyDeclaration := T_semi

This is an empty declaration, and therefore has the attribute value NULL.

ClassBodyDeclaration := { *Modifier* } *MemberDecl*

The synthesized attribute of this *ClassBodyDeclaration* is the result of applying each *Modifier* to the member declaration that is the attribute of *MemberDecl*.

Modifier := T_public

public is the default modifier for each class member declaration. A public member is accessible to all other classes.

Modifier := T_protected

A *protected* member of a class is only accessible from the class itself or derived classes.

Modifier := **T_private**

A **private** member of a class is only accessible from within the class itself.

Modifier := **T_static**

A **static** member of a class belongs to the class itself and not an instance of the class. Therefore, it can be accessed without creating an instance. A **static** method cannot use any non-static members or the **this** object.

Modifier := **T_final**

A **final** object is read-only and therefore, its value cannot be modified.

More than one modifier can be specified for a given member declaration without error. However, the following rule apply to the interpretation of a sequence of modifiers: when scanning the list of modifiers from left to right, an occurrence of a modifier from the set { **public**, **private**, **protected** } overrides all prior occurrences of any modifier from this set. Therefore, given the declaration

```
public private protected int mymember;
```

the member **mymember** is considered **protected**.

MemberDecl := *MethodDecl*

The synthesized attribute of *MemberDecl* is the attribute of *MethodDecl*, the declaration of a method.

MemberDecl := *Type* *VariableDeclarator* **T_semi**

The synthesized attribute of *MemberDecl* is the attribute of *VariableDeclarator*, a declaration of a member variable of type *Type*. This type must not be **void**, or an error must be reported.

MemberDecl := **T_identifier** *FormalParameters* *Block*

This rule is used to declare a constructor for the class. The identifier must be the name of the class, or an error is to be reported. The synthesized attribute of *MemberDecl* is the declaration of this special method.

VariableDeclarator := **T_identifier** { *BracketPair* } [*VariableInitializer*]

The synthesized attribute of *VariableDeclarator* is a declaration of a variable named by the identifier. Its type is obtained from an inherited attribute,

which is always a *Type*. Each *BracketPair* adds an array dimension. The *VariableInitializer*, if present, must be of a compatible type with the variable's type, or an error is to be reported.

VariableInitializer := *Expression*

The synthesized attribute of *VariableInitializer* is the type of the *Expression*.

VariableInitializer := *ArrayInitializer*

The synthesized attribute of *VariableInitializer* is the attribute of the *ArrayInitializer*.

ArrayInitializer := **T_lbrace** [*VariableInitializers*] **T_rbrace**

The synthesized attribute of *ArrayInitializer* is the attribute of the *VariableInitializers*.

VariableInitializers := *VariableInitializer* { **T_comma** *VariableInitializer* }

The synthesized attribute of *VariableInitializers* is a type obtained by applying an array type constructor to the type of the initial *VariableInitializer*. Each subsequent *VariableInitializer* must be of a compatible type with the initial *VariableInitializer*, or an error is to be reported.

MethodDecl := *Type* **T_identifier** *FormalParameters* *Block*

The synthesized attribute of *MethodDecl* is a declaration of the named method, with return type *Type* and argument types obtained from the attribute of *FormalParameters*. If a method by this name is already declared within this class, an error must be reported.

FormalParameters := **T_lpar** [*FormalParameterList*] **T_rpar**

The synthesized attribute of *FormalParameters* is the attribute of *FormalParameterList*, or NULL if there is no list.

FormalParameterList := *FormalParameter* { **T_comma** *FormalParameter* }

The synthesized attribute of *FormalParameterList* is a type obtained by applying the *list* constructor to types contained in the attributes of each *FormalParameters*, from left to right.

FormalParameter := [**T_final**] *Type* *VariableDeclaratorId*

The synthesized attribute of *FormalParameter* is the type obtained from

Type and any array dimensions obtained from *VariableDeclaratorId*. The declaration that is the attribute of *VariableDeclaratorId* is modified appropriately if **final** is present (thus inheriting the attribute of *FinalOpt*).

VariableDeclaratorId := **T_identifier** { *BracketPair* }

The inherited attribute of *VariableDeclaratorId* is a declaration of a new variable that is local to the enclosing method. It must not already be defined, or an error is to be reported. Its type is derived from the inherited *Type* of the *FormalParameter* that is the parent of this *VariableDeclaratorId* in the parse tree. Any *BracketPairs* are used to add array dimensions to the type of this declaration.

4.4 Statements

LocalVarDeclStmt := [**T_final**] *Type* *VariableDeclarators* **T_semi**

LocalVarDeclStmt has no attribute, but it must be verified that the *Type* is not **void**, or an error is to be reported. Furthermore, the attribute of *FinalOpt* must be inherited by *VariableDeclarators*, modifying each declaration to be **final** if necessary.

VariableDeclarators := *VariableDeclarator* { **T_comma** *VariableDeclarator* }

The synthesized attribute of *VariableDeclarators* is a list of variable declarations obtained from the synthesized attribute of each *VariableDeclarator*. *VariableDeclarators* also inherits an attribute from the production for *LocalVarDeclStmt*, the optional **final** modifier.

Statement := **T_if** *ParExpression* *Statement* **T_else** *Statement*

Statement := **T_if** *ParExpression* *Statement*

Statement := **T_while** *ParExpression* *Statement*

Statement := **T_do** *Statement* **T_while** *ParExpression* **T_semi**

In each case, the *ParExpression* must be of type **boolean**, or an error is to be reported.

Statement := **T_return** **T_semi**

If the enclosing method does not return type **void**, an error is to be

reported.

Statement := **T_return** *Expression* **T_semi**

A semantic error occurs if the attribute of **Expression** is not compatible with the return type of the enclosing method.

ParExpression := **T_lpar** *Expression* **T_rpar**

The synthesized attribute of *ParExpression* is that of the enclosed *Expression*.

4.5 Expressions

4.5.1 High-level Expressions

Expression := *Expression*₁ [**T_assign** *Expression*₂]

The synthesized attribute of *Expression* is the type of *Expression*₁. If this expression contains an assignment, then the type of *Expression*₂ must be compatible with that of *Expression*₁, or an error is to be reported.

*Expression*₁ := *Expression*₂ [**T_quest** *Expression*₁ **T_colon** *Expression*₂]

The synthesized attribute of *Expression*₁ is the type of *Expression*₂, if no ternary operator is present. Otherwise, the attribute is the type of *Expression*₁.

If the ternary operator is present, the type of *Expression*₂ must be **boolean**, or an error must be reported. Otherwise, the type of *Expression*₂ must be verified to be compatible with the type of *Expression*₁, or an error is reported.

4.5.2 Infix Operators

In all cases, the synthesized attribute of *Expression*₂ is the type of the result of the expression on the right side. An error must be reported if the operand types are invalid. In this case, the type of the result is **null**.

*Expression*₂ := *Expression*₂ **T_mulop** *Expression*₂

*Expression*₂ := *Expression*₂ **T_addop** *Expression*₂

The operand types must be numeric, and the result type is the highest ranked type among the operands.

$Expression2 := Expression2 \text{ T_relop } Expression2$

The operand types must be numeric, and the result type is **boolean**.

$Expression2 := Expression2_1 \text{ T_shifto } Expression2_2$

The operand types must be *integral*, or an error is to be reported. The result type is the type of $Expression2_1$.

$Expression2 := Expression2 \text{ T_eqop } Expression2$

The operand types must either both be *numeric*, both be **boolean**, or both be a *reference*. In the case of a *reference*, either operand may also be **null**. The result type is **boolean**.

$Expression2 := Expression2 \text{ T_and } Expression2$

$Expression2 := Expression2 \text{ T_or } Expression2$

The operand types and result type are all **boolean**.

$Expression2 := Expression2 \text{ T_bitand } Expression2$

$Expression2 := Expression2 \text{ T_bitor } Expression2$

The operand types must be *integral*. The result type is the highest ranked type among the operands.

$Expression2 := Expression2 \text{ T_instanceof } Expression2$

The operand type must be a *reference*. The result type is **boolean**.

$Expression2 := Expression3$

The synthesized attribute of $Expression2$ is the type that is the attribute of $Expression3$.

4.5.3 Prefix Operators

As with infix operators, if an invalid operand is supplied to a prefix operator, the type of the resulting expression is **null** and an error must be reported. The synthesized attribute of $Expression3$ is the type of the result of the operation.

$Expression3 := PrefixOp Expression3_1$

The synthesized attribute of $Expression3$ is the type of the result of applying $PrefixOp$ to $Expression3_1$.

Expression3 := *Expression4*

The synthesized attribute of *Expression3* is the synthesized attribute of *Expression4*.

PrefixOp := **T_incr** *PrefixOp* := **T_addop**

The operand must be *numeric*, and the result type is the same as the operand type.

PrefixOp := **T_not**

The operand must be *numeric* or **boolean**. The result type is the same as the operand type.

PrefixOp := **T_lpar** *Type* **T_rpar**

The operand type can be any type, and the result type is denoted by *Type*.

4.5.4 Postfix Operators

Expression4 := *Expression4₁* *PostfixOp*

The synthesized attribute of *Expression4* is the type of the result of applying *PostfixOp* to *Expression4₁*. This result type is stored as the synthesized attribute of *PostfixOp*. The *PostfixOp* inherits the attribute of *Expression4₁* for the purpose of computing its own attribute.

Expression4 := *Primary*

The synthesized attribute of *Expression4* is the attribute of *Primary*, the primary expression's type.

PostfixOp := **T_incr**

The attribute of *PostfixOp* is inherited from the attribute of the *Expression4* to which this operator is applied. The expression must have a *numeric* type or an error must be reported.

PostfixOp := *Arguments*

The type of the inherited expression must be a *method*, or an error must be reported. If it is a *method*, then the attribute of *Arguments*, a *list* of types, must be compatible with the given method's argument types, or an

error must be reported. The attribute of *PostfixOp* is the return type of the method. If the inherited expression is not a method, then the attribute of *PostfixOp* is the expression's type.

PostfixOp := *Selector*

The synthesized attribute of *PostfixOp* is the attribute of *Selector*.

Arguments := **T_lpar** [*ExpressionList*] **T_rpar**

The synthesized attribute of *Arguments* is the attribute of *ExpressionList*.

ExpressionList := *Expression* { **T_comma** *Expression* }

The synthesized attribute of *ExpressionList* is a type obtained by applying the *list* constructor to the types of each *Expression*.

Selector := **T_dot** *T_identifier*

Selector inherits the type of the expression to which this selector is being applied. To compute its own attribute, we proceed as follows: first, the inherited expression must be a reference; i.e. it must be an instance of some class. If not, an error is reported, and the *Selector* has type **null**. Otherwise, the identifier is looked up among the members of the class of which the inherited expression is an instance. If the member is not found, an error is reported, and the *Selector* has type **null**. Otherwise, the attribute of the *Selector* is the type of the member.

It also remains to check the access of the member. Let *C* be the class in which this member is found, and let *D* be the current class, i.e. the class containing the method in which this *Selector* appears. The following rules apply: If this member is **private**, then *C* and *D* must be the same class, or an error is reported. If this member is **protected**, then *D* must equal *C* or derive from *C*, or an error is reported.

Selector := **T_lsquare** *Expression* **T_rsquare**

Selector inherits the type of the expression to which this selector is being applied. This expression must be checked to determine whether it is a *creator*, i.e. an expression obtained by applying the **new** operator to some type. If it is, then the attribute of *Selector* is a new type obtained by applying the *array* constructor to the type of the expression, and thereby adding one array dimension.

If the inherited expression is not a creator, then it must be an array, or

an error is reported and the type of the expression itself is returned. If it is an array, then the attribute of *Selector* is the type obtained by removing one *array* constructor; i.e. removing one array dimension.

4.5.5 Primary Expressions

Primary := **T_lpar** *Expression* **T_rpar**

The synthesized attribute of *Primary* is the attribute of *Expression*.

Primary := **T_this**

The inherited attribute of *Primary* is the type of the enclosing class. If the enclosing method is **static**, an error must be reported.

Primary := *Literal*

The synthesized attribute of *Primary* is the type of the given *Literal*.

Primary := *QualifiedIdentifier*

The synthesized attribute of *Primary* is the type of the given *QualifiedIdentifier*.

Primary := **T_new** *Type*

The synthesized attribute of *Primary* is a new type obtained by applying the *create* constructor to *Type*.

Literal := **T_intliteral**

The synthesized attribute of *Literal* is the type **int**.

Literal := **T_floatliteral**

The synthesized attribute of *Literal* is the type **double**.

Literal := **T_charliteral**

The synthesized attribute of *Literal* is the type **char**.

Literal := **T_stringliteral**

The synthesized attribute of *Literal* is the type **char []**, i.e. the result of applying the *array* constructor to the **char** type.

Literal := **T_boolliteral**

The synthesized attribute of *Literal* is the type **boolean**.

Literal := **T_nullliteral**

The synthesized attribute of *Literal* is the type **null**.

QualifiedIdentifier := **T_identifier**

The identifier is looked up within the current scope, proceeding to enclosing scopes if necessary. If it is not found, an error is reported, and the inherited attribute of *QualifiedIdentifier* is the type **int**. Otherwise, the inherited attribute is the type recorded with the declaration that has been found. If this identifier refers to a member of a base class, it must not be a private member, or an error is to be reported.

QualifiedIdentifier := **T_sro ClassIdentifier T_dot T_identifier**

This rule is used to provide access to a static member of some class. The class denoted by *ClassIdentifier* is looked up, and if it is not found, an error must be reported and the attribute of *QualifiedIdentifier* is the type **int**.

If the named class is found, then the member named by **T_identifier** is looked up within that class. If it is not found, an error is reported, and the inherited attribute of *QualifiedIdentifier* is the type **int**.

If the member is found, then the inherited attribute of *QualifiedIdentifier* is the member's type. However, the modifiers of the member must be checked, in the order specified below. Let *C* be the class identified by the *ClassIdentifier*, and let *D* be the current class where this *QualifiedIdentifier* appears.

1. If the member is **private**, then *C* and *D* must be the same class, or an error is to be reported.
2. If the member is **protected**, then *D* must either equal *C*, or derive from *C*, or an error is to be reported.
3. If the member is not **static**, an error is to be reported.