The MINI Programming Language

(for CS321/322 compiler courses)

(*Version 1.4 Winter '10*)

Jingke Li
Department of Computer Science
Portland State University

1 Introduction

The **MINI** programming language is a small subset of Java, which supports classes and limited inheritance, simple data types, and a few structured control constructs.[1] This manual gives an informal definition for the language. Fragments of syntax are specified in BNF as needed; the complete grammar is attached as an appendix.

2 Lexical Issues

MINI is a subset of Java, hence it follows Java's lexical rules, but with some simplifications. Here are a few highlighted points:

- MINI is case sensitive — upper and lower-case letters are *not* considered equivalent.
- The following are MINI's *reserved* words — they must be written in the exact form as given:

  ```
  boolean class else extends false if int length main new public
  return static this true void while String System.out.println
  ```

  Note that `System.out.println` is treated as a single reserved word in MINI. This is to keep MINI compatible with Java, yet not worrying about supporting packages.
- *Identifiers* are strings of letters and digits starting with a letter, *excluding* the reserved keywords. Identifiers are limited to 255 characters in length.
- *Integers* contain only digits; they must be in the range 0 to $2^{31} - 1$. Note that an integer constant's value is always non-negative. However, an unary minus sign can be used to negate its value.
- *Strings* begin and end with a double quote (`"`) and contain any sequence of printable ASCII characters, except double quotes. Note in particular that strings may not contain tabs or newlines. String literals are limited to 255 characters in length, not including the delimiting double quotes.
- *Comments* can be in two forms: a single-line comment starts with `//` and ends with a newline character (`\n`); multi-line comments are enclosed in the pair `/*`, `*/`; they cannot be nested. Any character is legal in a comment.
- The following are MINI's remaining *operators* and *delimiters*:

  ```
  operator  = '='|'+'|'-'|'*'|'/'|"&&"|"||"|'!'|"=="|"!="|'<'|"<="|'>'|">="
  delimiter = ';'|','|'.'|'('|')'|'['|']'|'{'|'}'
  ```

[1] MINI is *not* the same as the Mini-Java language defined in Appel's text; MINI is slightly more powerful.

3 Program

A program is the unit of compilation for MINI. Each file read by the compiler must consist of exactly one program. There is no facility for linking multiple programs or for separate compilation of parts of a program. A program simply consists of a sequence of class declarations:

```
Program    -> ClassDecl {ClassDecl}
```

4 Classes

MINI supports inheritance. A class declaration can either define a base class or a subclass:

```
ClassDecl  -> "class" <ID> ["extends" <ID>] '{' {VarDecl} {MethodDecl} '}'
```

The body of a class consists of variable and method declarations. MINI requires that all variable declarations precede any method declaration. The class variables are dynamic, i.e. they are created for each object of the class. There is no *static* class variables in MINI.

All classes and their contents are public. A subclass inherits all contents of its parent. It may override a parent class's variables and/or methods. However, MINI *does not* support dynamic method binding. If there are multiple method definitions with the same name in both base and sub classes, MINI uses *static binding* to decide which one to use.

5 Methods

Method declarations have two syntax forms, a general form and a main-method form:

```
MethodDecl -> "public" Type <ID> '(' [Formals] ')' '{' {VarDecl} {Statement} '}'
           |  "public" "void" <ID> '(' [Formals] ')' '{' {VarDecl} {Statement} '}'
           |  "public" "static" "void" "main" '(' "String" '[' ']' <ID> ')'
                '{' {VarDecl} {Statement} '}'
Formals    -> Type <ID> {',' Type <ID>}
```

In the general form, a method declaration has a list of formal parameters (could be empty), a return type (could be `void`), and a body of variable declarations followed by statements. Methods declared in the same class share the same scope — hence they are treated as (potentially) mutually recursive.

A method may have zero or more *formal parameters*. Parameters are always passed by value. A method may have a return value of any type, in which case, the return statement(s) in the method body must return an expression of the corresponding type. A method may also be declared not to return any value (represented by the keyword `void`);[2] in this case the return statement(s) in the method body must *not* be accompanied with any expression. In general, there can be multiple return statements in a method body. There is an implicit `return` statement at the bottom of every method body.

Variable declared in a method are local to the method. Their declarations are not mutually recursive.

Main Method

Every MINI program should have a single *main method* declaration, which takes the following specific form:

```
public static void main (String[] <ID>) { ... }
```

Note that the sole parameter in the main method is considered a *dummy*, and cannot be used anywhere.

The class that includes the main method is called the *main class*. It must be static — no object can be created of it; also it should not have any variable of its own.

[2]Note that `void` is not a valid type in MINI's type system.

6 Variables

Variables may appear in two places in MINI: in the scope of a class declaration and in the scope of a method declaration. In both cases, MINI requires that variable declarations appear at the beginning of the scope, i.e. before any method declaration in the class case, and before any statement in the method case.

The syntax of variable declaration is simple:

```
VarDecl    -> Type <ID> ['=' FullExpr] ';'
```

Each declaration allows only one variable to be defined. A variable declaration may be initialized with a value, given by an expression (including new array/object allocation). Variable declarations take effect one at a time, in the written order; they are never recursive.

7 Types

MINI has three categories of types: *basic*, *object*, and *array*:

```
Type       -> ElmType ['[' ']']
ElmType    -> "boolean" | "int" | <ID>
```

Basic Types

There are two built-in basic types: Boolean and integer, represented by `boolean` and `int`, respectively. The Boolean type has two built-in values, `true` and `false`. It has no relation to the integer type: a Boolean value cannot be converted to or from an integer value.

Object Types

Class objects are of *object* types. They are represented by their corresponding class names.

Array Types

An array is a structure consisting of zero or more elements of the same type. Elements of an array must be of a type `boolean`, `int`, or an object type. (Consequently, nested arrays are not supported in in MINI.) An *array* type is specified by a type followed by a pair of square brackets. The elements of an array can be accessed by *dereferencing* using an *index*, which ranges from 0 to the length of the array minus 1. The length of an array is not fixed by its type, but is determined when the array is created at runtime. It is a checked runtime error to dereference outside the bounds of an array. A built-in method `.length()` can be invoked on any array object, and it returns the number of elements in the array.

Strong Typing Rules

MINI is a strongly-typed language; every expression has a unique type, and types must match at assignments, calls, etc.

8 Statements

Statement Block

```
Statement  -> '{' {Statement} '}'
```

Executing the sequence of statements in the given order.

Assignment

```
Statement  -> Lvalue '=' FullExpr ';'
```

The right-hand-side expression (including new array/object allocation) is evaluated and stored in the location specified by the left-hand-side. The lhs can be either a variable or an array element; the object that the variable or array belongs to may be explicitly specified.

Method Call

```
Statement  -> Lvalue '(' [Params] ')' ';'
Params     -> Expr {',' Expr}
```

This statement is executed by evaluating the argument expressions left-to-right to obtain actual parameter values, and then executing the proper method specified by `Lvalue` with its formal parameters bound to the actual parameter values until a `return` statement (with no expression) is executed.

If-then-else

```
Statement  -> "if" '(' Expr ')' Statement ["else" Statement]
```

This statement specifies the conditional execution of guarded statements. The guard expression must evaluate to a Boolean; if `true`, the 'then-clause' statement is executed; otherwise the 'else clause' statement is executed (if exists).

While

```
Statement  -> "while" '(' Expr ')' Statement
```

The statement is repeatedly executed as long as the expression evaluates to `true`.

Print

```
Statement  -> "System.out.println" '(' [PrintArg] ')' ';'
PrintArg   -> Expr | <STR>
```

Executing this statement writes the value of the specified expression (which must be of a basic type) or string to standard output, followed by a new line.

Return

```
Statement  -> "return" [Expr] ';'
```

Executing `return` terminates execution of the current method and returns control to the calling context. There can be multiple `return`s within one method body, and there is an implicit `return` at the bottom of every method. If a method requires a return value, then a `return` statement must specify a return value expression of the return type; otherwise it must not have an expression. The main method body must not include a `return`.

9 Expressions

Array and Object Allocation

```
FullExpr   -> "new" ElmType '[' <INT> ']'
           |  "new" <ID> '(' [Params] ')'
           |  Expr
```

New array and object allocation is a special form of expression. In MINI, it can only be used in variable initialization and assignment statement.

Binary and Unary Operations

```
Expr       -> Expr Binop Expr
           -> Unop Expr
Binop      -> '+' | '-' |  '*' | '/'
           -> "&&" | "||"
           -> "==" | "!=" | '<' | "<=" | '>' | ">="
Unop       -> '-' | '!'
```

Arithmetic Operators — These operators require integer arguments.

Logical Operators — These operators require Boolean operands and return a Boolean result. Both `&&` and `||` are "short-circuit" operators; they do not evaluate the right-hand operand if the result is determined by the left-hand one.

Relational Operators — These operators all return a Boolean result. They all work on integer arguments. Operators `==` and `!=` also work on pairs of Boolean arguments, or pairs of array or object arguments of the same type; in both cases, they test "pointer" equality (that is, whether two arrays or objects are the same instance, not whether they have the same contents). Note that relational expressions cannot be embedded into other expressions unless parenthesized. In other words, `a < b > c` is an illegal expression, while `(a < b) > c` is legal.

Simple Expressions

```
Expr       -> '(' Expr ')'
           -> Lvalue '(' [Params] ')'
           -> Lvalue '.' "length" '(' ')'
```

A simple expression denotes a parenthesized subexpress or the return value of a method call. Note that a method call is a valid expression only if the method has a return value. In particular, a method `length()` is defined on all array objects, and it returns the length of the array.

L-Values

```
Expr       -> Lvalue
Lvalue     -> [ "this" '.' ] <ID> {Deref}
Deref      -> '[' Expr ']' | '.' <ID>
```

An l-value is an expression denoting a location, whose value can both be read and assigned (hence it can appear on the left-hand-side of an assignment). The syntactic patterns of l-values include identifier, array element, and object member. Note that the index to an array must be of integer type. The first `<ID>` in the `Lvalue` clause refers to either a method or variable defined in the current scope (in this case, an optional `this` pointer could be used), or one that is inherited from a parent's scope.

Literals

```
Expr       -> Literal
Literal    -> <INT> | "true" | "false"
```

A literal expression evaluates to the literal value specified.

Associativity and Precedence

The arithmetic binary operators are all left-associative. The operators' precedence is defined by the following table:

highest	`new`, `()`
	`[]`, `.`(selector), method call
	`-`, `!`
	`*`, `/`
	`+`, `-`
	`==`, `!=`, `<`, `<=`, `>`, `>=`
	`&&`
lowest	`||`

A Complete MINI Syntax

```
Program    -> ClassDecl {ClassDecl}
ClassDecl  -> "class" <ID> ["extends" <ID>] '{' {VarDecl} {MethodDecl} '}'
MethodDecl -> "public" Type <ID> '(' [Formals] ')' '{' {VarDecl} {Statement} '}'
           |  "public" "void" <ID> '(' [Formals] ')' '{' {VarDecl} {Statement} '}'
           |  "public" "static" "void" "main" '(' "String" '[' ']' <ID> ')'
                '{' {VarDecl} {Statement} '}'
Formals    -> Type <ID> {',' Type <ID>}
VarDecl    -> Type <ID> ['=' FullExpr] ';'
Type       -> ElmType ['[' ']']
ElmType    -> "boolean" | "int" | <ID>
Statement  -> '{' {Statement} '}'
           |  Lvalue '=' FullExpr ';'
           |  Lvalue '(' [Params] ')' ';'
           |  "if" '(' Expr ')' Statement ["else" Statement]
           |  "while" '(' Expr ')' Statement
           |  "System.out.println" '(' [PrintArg] ')' ';'
           |  "return" [Expr] ';'
Params     -> Expr {',' Expr}
PrintArg   -> Expr | <STR>
FullExpr   -> "new" ElmType '[' <INT> ']'
           |  "new" <ID> '(' [Params] ')'
           |  Expr
Expr       -> Expr Binop Expr
           |  Unop Expr
           |  '(' Expr ')'
           |  Lvalue '(' [Params] ')'
           |  Lvalue '.' "length" '(' ')'
           |  Lvalue
           |  Literal
Lvalue     -> [ "this" '.' ] <ID> {Deref}
Deref      -> '[' Expr ']' | '.' <ID>
Literal    -> <INT> | "true" | "false"
Binop      -> '+' | '-' | '*' | '/' | "&&" | "||" | "==" | "!=" | '<' | "<=" | '>' | ">="
Unop       -> '-' | '!'
```

B Abstract Syntax Tree Nodes

Here is the official abstract syntax format for MINI.

```
Program
  ClassDeclList cl

*** declarations ***

ClassDecl
  Id cid, pid
  VarDeclList vl
  MetDeclList ml

VarDecl
  Type t
  Id var
  Exp e

MethodDecl
  Type t
  Id mid
  FormalList fl
  VarDeclList vl
  StmtList sl

Formal
  Type t
  Id id

*** types ***

IntType

BoolType

ArrayType
  Type et

ObjType
  Id cid

*** statements ***

Block
  StmtList sl

Assign
  Exp lhs, rhs

CallStmt
  Exp obj
  Id mid
  ExpList args

If
  Exp e
  Stmt s1, s2

While
  Exp e
  Stmt s

Print
  Exp e

Return
  Exp e

*** expressions ***

NewArray
  Type et
  Int size

NewObj
  Id cid
  ExpList args

Binop
  int op      // ADD, SUB, MUL, DIV, AND, OR
  Exp e1, e2

Relop
  int op      // EQ, NE, LT, LE, GT, GE
  Exp e1, e2

Neg
  Exp e

Not
  Exp e

ArrayElm
  Exp array, idx

ArrayLen
  Exp array

Call
  Exp obj
  Id mid
  ExpList args

Member
  Exp obj
  Id var

This

Id
  String s

Int
  int i

Bool
  int b

Text
  String s
```