The MINI Programming Language
(for CS321/322 compiler courses)

Jingke Li
Dept. of Computer Science
Portland State University

(*version 1.1 w'06*)

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

  Note that `System.out.println` is treated as a 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.
- *Constants* are either integer, real, or string. *Integers* contain only digits; they must be in the range 0 to $2^{31}-1$. *Reals* contain a decimal point; a digit is required before the decimal point, but *not* afterwards. Note that neither an integer nor a real can be negative, since there is no provision for a minus sign. *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 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> '(' [FormalParams] ')'
                  '{' {VarDecl} {Statement} '}'
             -> "public" "static" "void" "main" '(' "String" '[' ']' <ID> ')'
                  '{' {VarDecl} {Statement} '}'
FormalParams -> Formal {',' Formal}
Formal       -> 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 `void` type); 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.

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> ['=' Expr] ';'
```

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

7 Types

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

```
Type         -> BasicType ['[' ']'] | <ID> | "void"
BasicType    -> "boolean" | "int" | "double"
```

Basic Types

There are three built-in *basic* types: `boolean`, `int`, and `double`. Integer constants all have type `integer`, real constants all have type `double`, and the built-in values `true` and `false` have type `boolean`.

The `int` and `double` types collectively form the *numeric* types. An `int` value will always be explicitly *coerced* to a `double` value if necessary. The boolean type has no relation to the numeric types, and a boolean value cannot be converted to or from a numeric value.

Array Types

An array is a structure consisting of zero or more elements of the same basic type. An *array* type is represented by a basic 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 build-in method `.length()` can be invoked on any array object, and it returns the number of elements in the array.

Object Types

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

Void Type

A special *void* type can be used (only) to specify the return type of a method. It indicates that the method does not return any value.

Strong Typing Rules

MINI is a strongly-typed language; every expression has a unique type, and types must match at assignments, calls, etc. (except that an integer can be used where a real is expected.)

8 Statements

Statement Block

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

Executing the sequence of statements in the given order.

Assignment

```
Statement    -> [Expr '.'] <ID> ['[' Expr ']'] '=' Expr ';'
```

The rhs expression is evaluated and stored in the location specified by the lhs. 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    -> [Expr '.'] <ID> '(' [ExpList] ')' ';'
ExpList      -> 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 `[Expr '.']  <ID>` 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" '(' [Expr|<STRING>] ')' ';'
```

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 `returns` 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

Simple Expressions

```
Expr        -> "new" BasicType '[' Expr ']'
            -> "new" <ID> '(' ')'
            -> '(' Expr ')'
            -> "this"
            -> Number
Number      -> <INT> | <REAL> | "true" | "false"
```

A simple expression is a *new* array or class object, subexpression with parentheses, `this` pointer, or a number. A number expression evaluates to the literal value specified. Note that reals are distinguished from integers by lexical criteria (see Section 2).

Array Elements and Object Members

```
Expr        -> Expr '[' Expr ']'
            -> Expr '.' "length" '(' ')'
            -> [Expr '.'] <ID> '(' [ExpList] ')'
            -> [Expr '.'] <ID>
```

The index to an array must be of integer type. A method `length()` is defined on all array objects, and it returns the length of the array. Object member variables have the syntax `[Expr '.']  <ID>`. If the object expression is omitted, then the `<ID>` is either defined in the current scope or is inherited from a parent's scope. A method call is a valid expression only if the method has a return value.

Arithmetic Operators

```
Expr        -> Expr Binop Expr
Binop       -> '+' | '-' |  '*' | '/'
```

These operators require numeric arguments. If both arguments are integers, an integer operation is performed and the integer result is returned; otherwise, any integer arguments are coerced to reals, a real operation is performed, and the real result is returned.

Logical Operators

```
Expr        -> Expr Binop Expr
            -> '!' Expr
Binop       -> "&&" | "||"
```

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

```
Expr        -> Expr Binop Expr
Binop       -> "==" | "!=" | '<' | "<=" | '>' | ">="
```

These operators all return a boolean result. They all work on numeric arguments; if both arguments are integer, an integer comparison is made; otherwise, any integer argument is coerced to real and a real comparison is made. 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).

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.

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} '}'
VarDecl      -> Type <ID> ['=' Expr] ';'
MethodDecl   -> "public" Type <ID> '(' [FormalParams] ')'
                  '{' {VarDecl} {Statement} '}'
             -> "public" "static" "void" "main" '(' "String" '[' ']' <ID> ')'
                  '{' {VarDecl} {Statement} '}'
FormalParams -> Formal {',' Formal}
Formal       -> Type <ID>
Type         -> BasicType ['[' ']'] | <ID> | "void"
BasicType    -> "boolean" | "int" | "double"
Statement    -> '{' {Statement} '}'
             -> [Expr '.'] <ID> ['[' Expr ']'] '=' Expr ';'
             -> [Expr '.'] <ID> '(' [ExpList] ')' ';'
             -> "if" '(' Expr ')' Statement ["else" Statement]
             -> "while" '(' Expr ')' Statement
             -> "System.out.println" '(' [Expr|<STRING>] ')' ';'
             -> "return" [Expr] ';'
Expr         -> Expr Binop Expr
             -> '!' Expr
             -> Expr '[' Expr ']'
             -> Expr '.' "length" '(' ')'
             -> [Expr '.'] <ID> '(' [ExpList] ')'
             -> [Expr '.'] <ID>
             -> "new" BasicType '[' Expr ']'
             -> "new" <ID> '(' ')'
             -> '(' Expr ')'
             -> "this"
             -> Number
ExpList      -> Expr {',' Expr}
Number       -> <INT> | <REAL> | "true" | "false"
Binop        -> '+' | '-' |  '*' | '/' | "&&" | "||"
             -> "==" | "!=" | '<' | "<=" | '>' | ">="
```

B Abstract Syntax Tree Nodes

Here is the official communication format for the abstract syntax for MINI.

```
Program
  ClassDeclList

ClassDecl
  Id -- class id
  Id -- parent class id
  VarDeclList
  MetDeclList

VarDecl
  Type
  Id
  Exp

MetDecl
  Type
  Id -- method id
  FormalList
  VarDeclList
  StmtList

Formal
  Type
  Id

Type -> BasicType | ArrayType
      | ObjType | VoidType

BasicType
  typ -- BOOL, INT, or REAL

ArrayType
  BasicType -- element type

ObjType
  Id -- class id

Stmt -> Block | Assign | CallStmt
      | If | While | Print | Return
      | StmtList

Block
  StmtList

Assign
  Exp -- lhs
  Exp -- rhs

CallStmt
  Exp -- object
  (Id -- class id)
  Id -- method id
  ExpList -- arguments

If
  Exp
  Stmt -- then clause
  Stmt -- else clause

While
  Exp
  Stmt

Print
  Exp

Return
  Exp

Exp -> Binop | Relop | Not | ArrayElm
     | ArrayLen | Call | NewArray
     | NewObject | Int | Real | Text
     | This | True | False | ExpList

Binop
  op -- ADD, SUB, MUL, DIV, AND, OR
  Exp
  Exp

Relop
  op -- EQ, NE, LT, LE, GT, GE
  Exp
  Exp

Not
  Exp

ArrayElm
  Exp -- array
  Exp -- idx
  (BasicType -- element type)

ArrayLen
  Exp -- array

Call
  Exp -- object
  (Id -- class id)
  Id -- method id
  ExpList -- arguments

NewArray
  BasicType -- element type
  Exp -- size

NewObject
  Id -- class id
```