Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue 10:00-11:50 @SRTC 155, Labs: Tue 14:00-15:50 & Wed 12:00-13:50 @FAB 88-10

The miniJava Programming Language

1 Introduction

The **miniJava** programming language is a small subset of Java. It supports classes and inheritance, simple data types, and a few structured control constructs. 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 Rules

The miniJava language's token definitions follow Java's lexical rules in most cases; but miniJava differs from Java in some places (for simplification purpose).

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

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

  Note that the words in the second row are not reserved in Java. They are made reserved in miniJava to simplify compatibility with Java. (For example, you can use `System.out.println` to print in miniJava even though miniJava does not support packages.)
- *Identifiers* are strings of letters and digits starting with a letter (not to include the reserved keywords). There is no length limit.
- *Integer* literals contain only digits; their values must be in the range 0 to $2^{31}-1$. Note that an integer literal's value is always non-negative. To get a negative integer value, an unary minus operator can be used.
- *String* literals begin and end with a double quote (`"`) and contain any sequence of ASCII characters, except double quotes (`"`), carriage returns (`\r`), and newlines (`\n`). A string can be of arbitrary length, including zero. Note that the beginning and ending quotes are *not* part of a string literal; they are delimiters. For instance, the string `"1"` is of length one, not three.
- *Comments* can be in two forms: a single-line comment starts with `//` and ends with a (invisible) newline character (`\n`); multi-line comments are enclosed in the pair `/*`, `*/`; they cannot be nested. All ASCII characters are legal in a comment.
- The following are miniJava's *operators* and remaining *delimiters*:

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

There are no other lexical entities in miniJava. Many Java lexical features are not supported in miniJava: for example, there are no floating point literals or hexadecimal numbers.

3 Program

A program is the unit of compilation for miniJava. 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.

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

4 Classes

Since miniJava 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. miniJava 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 miniJava. Note that the class contains the "`main`" method is automatically a static class by Java's rule. Hence there cannot be any class variables in that class in a miniJava program.

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. If there are multiple method definitions with the same name in both base and sub classes, miniJava uses *dynamic binding* to decide which one to use.

Note that miniJava does not support explicit object constructors. When a new class object is created, a default constructor will be called. (See "Array and Object Allocation" in Section 9 for more information on this.)

5 Methods

A general 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. The special "main" method has a sole parameter, which is considered a *dummy*, and cannot be used anywhere:

```
MethodDecl -> "public" ExtType <ID> "(" [Param {"," Param}] ")"
                "{" {VarDecl} {Stmt} "}"
            | "public" "static" "void" "main" "(" "String" "[" "]" <ID> ")"
                "{" {VarDecl} {Stmt} "}"
ExtType    -> Type | "void"
Param      -> Type <ID>
```

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`); 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.

6 Variables

Variables may appear in two places in miniJava: in the scope of a class declaration and in the scope of a method declaration. In both cases, miniJava 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> ["=" InitExpr] ";"
```

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

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

```
Type       -> BasicType
           |  BasicType "[" "]"      // array type
           |  <ID>                   // object type
BasicType  -> "int" | "boolean"
```

Basic Types

There are two built-in basic types: `int` and `boolean`. 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.

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 basic type. (Consequently, nested arrays are not supported in miniJava.) 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 array-length − 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.

Object Types

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

Strong Typing Rules

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

8 Statements

Statement Block

```
  Stmt -> "{" {Statement} "}"
```

Executing the sequence of statements in the given order.

Assignment

```
  Stmt -> Lvalue "=" InitExpr ";"
```

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

```
  Stmt -> ExtId "(" [Args] ")" ";"
  Args -> 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

```
  Stmt -> "if" "(" Expr ")" Stmt ["else" Stmt]
```

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

```
  Stmt -> "while" "(" Expr ")" Stmt
```

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

Print

```
  Stmt     -> "System" "." "out" "." "println" "(" [PrintArg] ")" ";"
  PrintArg -> Expr | <STRLIT>
```

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

```
Stmt -> "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

```
InitExpr -> "new" BasicType "[" <INTLIT> "]"
         |  "new" <ID> "(" ")"
         |  Expr
```

These two expressions are for creating a new array and a new object in the heap space, respectively. When a new array is allocated, all its elements are initialized to the default value of its element type per Java's rule — *i.e.*, an integer array will be initialized to 0s, and a boolean array will be initialized to `false`.

Note that the new object expression does not take any argument. When a new object is created, a default constructor will be executed, which will initialize the object's fields with either their static initial values (if exist from program), or Java's default values (*i.e.*, integer field will be initialized to 0, boolean field to `false`, and object field to `null`).

In miniJava, these two expressions can only be used in variable initialization and assignment statement. In other words, they can not be embedded in other expressions.

Binary and Unary Operations

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

Arithmetic Operators — These operators require integer operands.

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 operands. Operators == and `!=` also work on pairs of boolean operands, or pairs of array or object operands 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).

Simple Expressions

```
Expr -> "(" Expr ")"
     |  ExtId "(" [Args] ")"
```

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.

L-Values

```
Expr   -> Lvalue
Lvalue -> ExtId "[" Expr "]"            // array element
       |  ExtId
ExtId  -> ["this" "."] <ID> {"." <ID>}  // object field or just 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, object field, and array element. Note that the index to an array must be of integer type. The first `<ID>` in the `ExtId` 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 -> <INTLIT> | "true" | "false"
```

A literal expression evaluates to the literal value specified.

Associativity and Precedence

The arithmetic and logical binary operators are all left-associative. The operators' precedence is defined by the following list (from high to low):

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

A Complete miniJava Syntax

```
Program    -> {ClassDecl}

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

MethodDecl -> "public" ExtType <ID> "(" [Param {"," Param}] ")"
                "{" {VarDecl} {Stmt} "}"
           |  "public" "static" "void" "main" "(" "String" "[" "]" <ID> ")"
                "{" {VarDecl} {Stmt} "}"

Param      -> Type <ID>

VarDecl    -> Type <ID> ["=" InitExpr] ";"

ExtType    -> Type | "void"

Type       -> BasicType
           |  BasicType "[" "]"             // array type
           |  <ID>                          // object type

BasicType  -> "int" | "boolean"

Stmt  ->      "{" {Stmt} "}"                      // stmt block
           |  ExtId "(" [Args] ")" ";"            // call stmt
           |  Lvalue "=" InitExpr ";"             // assignment
           |  "if" "(" Expr ")" Stmt ["else" Stmt]
           |  "while" "(" Expr ")" Stmt
           |  "System" "." "out" "." "println"
                "(" [PrintArg] ")" ";"
           |  "return" [Expr] ";"

Args       -> Expr {"," Expr}

PrintArg   -> Expr | <STRLIT>

InitExpr   -> "new" BasicType "[" <INTLIT> "]"    // new array
           |  "new" <ID> "(" ")"                  // new object
           |  Expr

Expr       -> Expr BinOp Expr
           |  UnOp Expr
           |  "(" Expr ")"
           |  ExtId "(" [Args] ")"          // method call
           |  Lvalue
           |  Literal

Lvalue     -> ExtId "[" Expr "]"            // array element
           |  ExtId

ExtId      -> ["this" "."] <ID> {"." <ID>}  // object field or just ID

Literal    -> <INTLIT> | "true" | "false"

BinOp      -> "+" | "-" | "*" | "/" | "&&" | "||" | "==" | "!=" | "<" | "<=" | ">" | ">="
UnOp       -> "-" | "!"
```