A Short Summary of Javali

Thomas Gross

December 9, 2011

1 Introduction

Javali is a simple language based on ideas found in languages like C++ or Java. Its purpose is to serve as the source language for a simple compiler project. The goal is to expose the student to the core problems that a compiler for an object-oriented language must address. Extensions or variants of the language allow a student to explore a variety of topics in the domain of compiler construction.

This short text attempts summarize the important features of Javali. For those familiar with Java, it is probably enough to read the following section.

2 Differences from Java

Because Javali is mostly a subset of Java, this section quickly covers the main differences:

- Input and output are performed using the built-in functions, `read()`, `write()`, and `writeln()`.
- No string constants or floating point values.
- No static methods or fields.
- All classes are contained in a single file.
- Program executing begins in a class called `Main` with a method `main()`.
- There are no non-void methods.
- No constructors, generic typing, interfaces, or abstract methods/classes.
- Some statements, such as `switch` and `for`, have been removed. (see the grammar in Appendix A)
- Arrays are not co-variant. In other words, the supertype of all arrays is `Object`.
- As in Perl, `if` and `while` require braces {} around the conditional statements / loop bodies.
- In method bodies, all variables must be declared before the first statement.

During the course of the semester, you may lift some of the above restrictions in exchange for extra credit on the assignments.

3 Overall program structure

Javali uses a small set of reserved words. These keywords must be in all lowercase.

```
boolean    false      int      true
class      new        void     null
while      else       read     write
extends    if         this     writeln
```

Programmer-defined identifiers (Ident) for variables, types, methods (functions), etc. follow the conventions of many programming languages (start with a letter etc). Javali is case sensitive.
A Javali program consists of a number of units. These units are called classes.
A file can contain the definitions of multiple classes. There is no model of a linker or a library, so the complete program must be contained in a single file.
The definition of each class contains declarations of variables, and methods. Every Javali program must include a class named `Main`. This `Main` class must contain a method called `main()` which takes no parameters, from which execution will begin.
Note that the grammar allows each method to contain nested definitions of methods. Support for such methods is optional, however.

4 Variables and types

A variable of type `int` is represented by 32 bits. Variables of type `boolean` can be represented by 1 bit, 1 byte, or 32 bits, depending on the particulars of your implementation; often it is simplest to use the same representation as an `int`.

Javali includes also reference types. A variable with a reference type allows access to either an array or to an instance of some class. Reference variables are typed, i.e. a reference variable can only be used to access either arrays (with elements of some type) or instances of a specific class.

The definition of a reference variable for arrays must include only the type of the elements.

The actual size of an array can be controlled at runtime. There are no restrictions on the elements of an array; they can be of a basic type or can be of a reference type.

Classes contain data fields (or data members); there are no restrictions on the types of the field of a class.

4.1 Types

New Javali types are defined by defining classes. Each class has a name (given by an identifier), and within each scope, the name of a class must be unique.

4.2 Scoping

We distinguish between the *scope* of a variable and its *extent*. The scoping rules determine the visibility of a named item (variable, method, class); the extent determines the lifetime of a variable.

Javali uses lexical scoping: a reference to a type, variable, or method always refers to the corresponding item that is defined in the most interior enclosing scope. Definitions in an interior scope with identifier `ident` hide any definitions in an outer scope that use the identical identifier `ident`. It is illegal for a program to contain naming conflicts: the name spaces created by the definitions of variables, types, and methods within a scope cannot overlap.

There are several levels of static scoping. The outermost static scope is the class, which contains the names of all of its fields. In addition, there is a separate scope for each superclass with the names of the fields inherited from those superclasses. Next comes a scope for each method, which contains the method's parameters and any local variables. It is illegal to declare two identifiers with the same name in any scope. Therefore, local variables and parameters may not shadow one another, but they may shadow fields. Likewise, no class may contain two definitions for the same field, but a field in a subclass may shadow a field in a superclass. These are intended to be the same rules that Java uses.

A variable is visible from the point it is defined through the end of the scope it is defined in (the syntactic end of the method or class the variable or type is defined in). A type is visible in the scope it is defined in (and all enclosed scopes, unless hidden by another type definition). That is, a "forward reference" is allowed. Here is an example of a legal definition:

```
class Y {
  X xref;
}

class X {
  int i;
}

class Z {
  Z xref;
}
```

All variables and types are statically defined.

At runtime, a variable starts to exist when the scope that it is defined in is instantiated. For a class this is the time a new instance of this class is instantiated with the `new` operator. For a method this is the moment the method is invoked (called) by some other part of a program. Each invocation of a method creates a new dataspace. For arrays this is the time a new datablock is allocated with the `new` operator.

There are no rules or restrictions about the initial data values of variables. The behavior of programs which read variables (either fields or local variables) that have not been written is undefined.

Variables of a basic type are referred to by their identifier. Elements of an array or fields of a class instance are referred to using the standard "dot" notation.

The elements of an array are numbered starting with 0. Accesses to array elements that are not defined constitute an error, and result in undefined behavior.

It is an error to access a variable that is not defined. It is also an error to access a data field that is not defined. The compiler must flag such errors.

5 Methods

Methods can have an arbitrary number of parameters. If the parameter is of a basic type, call-by-value is used. If the parameter is a reference type, call-by-reference is used. There is no overloading, and it is illegal to pass the same reference variable more than once as an actual parameter for a given call site.

The type of an actual parameter must be identical to or a subtype of the type of the formal parameter.

5.1 Built-in methods

There are three build-in methods for basic I/O:

- `read()` reads a single integer from `stdin`
- `write(expr)` evaluates its argument, which must be of type `int`, and writes the resulting integer value to `stdout`
- `writeln()` writes a newline to `stdout`

These methods are defined in all scopes. It is an error to hide these methods (therefore, the names of these methods are included in the list of reserved keywords).

6 Statements

The usual rules and expectations exist for arithmetic operators, the assignment statement, the conditional statement, function call, and the loop constructs.

The two sides of an assignment statement must have identical types, or the right-hand side's type must be a subtype of the left-hand side's type. Assignment is by copying of values (for basic types) or by copying of the reference (for reference types).

7 Expressions

Expressions in Javali are evaluated like their Java equivalents.

Arithmetic expressions can be evaluated in any order that does not violate the usual semantics of the operators. The compiler is free to reorder statements as long as the program semantics are preserved.

It is not required to implement the boolean operators (like `||` or `&&`) in a short-circuited fashion, although if you do so it will count for extra credit.

8 Comments

As in C++ or Java, comments are either enclosed in `/*` and `*/`, or begin with `//` and extend to the end of the line. Comments cannot be nested.

9 Compilation errors

The compiler can abandon a translation as soon as an error is discovered in a source program. The compiler should provide a meaningful error message, but if the compiler cannot identify the the exact position in the program that is responsible for the error, it is acceptable to terminate the compilation with the error message "syntax error". It is, however, desirable that the compiler reports the line number of at least one item or construct that is involved in the error.

If there are multiple errors, the compiler is free to abandon compilation as soon as a single error is discovered.

10 Runtime system

The `new` operator allows the creation of instances of classes and to define storage for arrays. The operand of a `new` operator is a qualified type, i.e. either the name of a defined type (in this case, an instance of this class is created), or the name of a type followed by `[` *expr* `]`, where *expr* evaluates to an integer value. This value determines the size of the array.

Arrays and class instances are unintialized after creation; the programmer is responsible for proper initialization.

```
  // reference to int array int [] iarray; // creates int array iarray
 = new int[10]; // init of element iarray[0] = 0;

  // reference to array of Structs
  class Struct { int s; }
  Struct [] sref;
  // creates array of Struct references
  sref = new Struct[10];
  // init of element
  sref[0] = new Struct();
  // init of record
  sref[0].s = 127;
```

The `new` operator returns a reference to the newly created array or class instance. This is the only way to produce a reference to an array or instance.

11 Inheritance

The base version of Javali supports only single inheritance. The keyword `extends` expresses that a class `A` is a subclass of another class `Base`.

```
class Base {
 ...
}

class A extends Base {
 ...
}
```

The rule for ClassDecl states that extension of a class is optional. Subclasses inherit all methods and fields of their superclass(es). Fields can be shadowed and methods can be overriden. Overloading of method names is not permitted (as a consequence, methods inherited from a superclass can be overriden but not overloaded). This restriction is imposed to ease the implementation of the compiler.

If a method is invoked, then the target of the method invocation is determined by the actual type of the object instance (and not by the static type of the reference variable).

All instances of a subclass are also instances of the superclass. It is therefore always possible to assign a reference to a subclass instance to a supertype reference variable.

```
Base bref;
A aref;

...

bref = aref;
```

The special class `Object` is the superclass of all other classes, and of arrays. `Object` has no methods or fields, it is defined implicitly and it is illegal for a program to define this class explicitly.

`null` is compatible with all reference types.

When a reference `bref` (with some static type `Base` refers to an instance of a subtype `A` of `Base`, then an assignment of `bref` to a reference variable `aref` of type `A` requires a *cast*.

```
aref = (A) bref;
```

Such casts should be checked by the compiler and, if necessary, the runtime. On the other hand, upcasts (i.e., casts from a sub-type to a base type) are optional and require no runtime check.

11.1 Restrictions, simplifications and extensions

One option to simplify the compiler development is to ignore cast operations. That is, the check required for a downcast (as in the example above) is suppressed. This approach is taken by some languages and the assignment will state if you can make this simplification. But extra credit may be given to those who implement the checks.

An extension is to suppress the checks only when the compiler can determine that the check is unnecessary.

To simplify the implementation, the compiler can treat arrays of a class `Base` different from an array of class `A`, with `A` a subclass of `Base`. Consider this scenario:

```
class Base { ... }
class A extends Base { ... ]

A[] arefArray;
Base[] brefArray;
```

In Java, `A[]` is considered a subtype of `Base[]`. Javali treats these two differently to allow a simpler compiler implementation.

Extra credit for those implementations that *do not* have this restriction. (Hint: if you want to remove the restriction, any assignment to `arefArray[index]` must be checked that indeed a reference to an instance of `A` is stored in the array.)

A Syntax

Note: A | indicates a choice, a pair of curly brackets { and } indicates zero, one or more repetitions, and a pair of squared brackets [and] indicates zero or one repetition.

Ident ::= Letter { Letter | Digit } .
Letter ::= *letters* “A” *to* “Z” | “a” *to* “z” .
Boolean ::= “true” | “false” .
Integer ::= Digit { Digit } | “0x” HexDigit { HexDigit } .
Digit ::= *digits* “0” *to* “9” .
HexDigit ::= Digit | *letters* “a” *to* “f” | *letters* “A” *to* “F”.
PrimitiveType ::= “int” | “boolean” .
Type ::= PrimitiveType | Ident | ArrayType .
ArrayType ::= Ident “[” “]” | PrimitiveType “[” “]” .
QualifiedType ::= Ident “(” “)” | Ident ElemSelector | PrimitiveType ElemSelector .
SelectorSeq ::= { FieldSelector | ElemSelector } .
FieldSelector ::= “.” Ident .
ElemSelector ::= “[” SimpleExpr “]” .
NoSignFactor ::= IdentAccess | Integer | Boolean | “(” Expr “)”
| “!” Factor | “(” RefType “)” NoSignFactor | “null” .
Factor ::= [“+” | “-”] NoSignFactor .
RefType ::= Ident | Ident “[” “]” | “int” “[” “]” | “boolean” “[” “]”.
Term ::= Factor { (“*” | “/” | “%” | “&&”) Factor } .
SimpleExpr ::= Term { (“+” | “-” | “||”) Term } .
Expr ::= SimpleExpr [(“==” | “!=” | “<” | “<=” | “>” | “>=”) SimpleExpr] .
IdentAccess ::= (Ident | “this”) SelectorSeq .
Assignment ::= IdentAccess “=” (Expr | “new” QualifiedType | “read” “(” “)”) “;” .
MethodCall ::= (IdentAccess | “write” | “writeln”) “(” [ActualParam] “)” “;” .
ActualParam ::= Expr { “,” Expr } .
IfStmt ::= “if” “(” Expr “)” StmtBlock [“else” StmtBlock] .
WhileStmt ::= “while” “(” Expr “)” StmtBlock .
Stmt ::= Assignment | MethodCall | IfStmt | WhileStmt .
StmtBlock ::= “{” { Stmt } “}” .
IdentList ::= Ident { “,” Ident } .
FormalParam ::= Type Ident { “,” Type Ident } .
MethodDecl ::= MethodHeading MethodBody .
MethodHeading ::= “void” Ident “(” [FormalParam] “)” .
MethodBody ::= “{” DeclSeq { Stmt } “}” .
VarDecl ::= Type IdentList “;” .
DeclSeq ::= { VarDecl | MethodDecl } .
ClassDecl ::= “class” Ident [“extends” Ident] “{” DeclSeq “}” .
Unit ::= ClassDecl { ClassDecl } .

B Example program: Quicksort

Here is an example program, without warranty.

```
class Record {
    int a ;
}

class Main {
    Record [] a;
    int i;

    void swap(Record r1, Record r2) {
        int temp;

        temp = r1.a;
        r1.a = r2.a;
        r2.a = temp;
    }

    void sort(int left, int right) {
        int i,j;
        int m;

        m = (a[left].a + a[right].a) / 2;
        i = left;
        j = right;
        while (i <= j) {
            while (a[i].a < m) { i = i+1; }
            while (a[j].a > m) { j = j-1; }
            if (i <= j) {
                swap(a[i], a[j]);
                i = i+1;
                j = j-1;
            }
        }
        if (left < j) { sort(left,j); }
        if (i < right) { sort(i,right); }
    }

    void main() {
        int SIZE;
        int j;

        SIZE = 5;
        a = new Record[SIZE];
        j = 0;
        while (j<SIZE) {
          a[j] = new Record();
          j = j + 1;
```

```
        }
        a[0].a = 5; a[1].a = 3; a[2].a = 1; a[3].a = 4; a[4].a = 2;
        sort(0,4);
        i = a[3].a;
        write(i);
        writeln();
    }
}
```