A Short Summary of Javali

October 15, 2015

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 the 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.
- No static methods or static fields.
- No casts between primitive types.
- All classes are contained in a single file.
- Program execution begins in a class called `Main` with a method `main()` with return type `void` and no parameters.
- 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`.

- `if` and `while` require braces {} around the conditional statements / loop bodies.
- In method bodies, all variables must be declared before the first statement.

3 Program structure

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

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

Programmer-defined identifiers for variables, types, methods (functions), etc. follow the conventions the Java programming language (start with a letter etc). Javali is case sensitive.
A Javali program consists of multiple units that are called classes. The definition of each class contains declarations of instance variables, and methods.
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.
Every Javali program must include a class named `Main`. This `Main` class must contain a method called `main()`, which takes no parameters and has `void` as its return type, from which execution will begin.

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 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), the name of a class must be unique.

4.2 Scoping

In the following, we use the term *name* to refer to the word in the program text, e.g., a type, a variable, a method, etc. And we use the term *symbol* to refer to the entity that describes semantic information about the *name*, such as the type, if it is a variable or a field, if it is a type or a variable, etc.

Scoping defines the way that names are resolved to symbols. Javali uses lexical scoping: a name in the program can always be resolved statically to a symbol based on the program text. Generally, names refer to the symbols in the innermost enclosing scope. Symbols in an interior scope hide any symbols with the same name of the outer scopes. For example, a local variable hides a field of the same name. There must be no two symbols with the same name in the same scope. In this case there is a *naming conflict*, e.g., two local variables with the same name.

There are three levels of static scoping in Javali, from outermost to innermost:

1. **Global scope:** contains symbols defined on the file level, such as classes or built-in types
2. **Class scope:** contains symbols defined on the class level, such as fields and methods (incl. those of the super-class)
3. **Method scope:** contains symbols defined on the method level, such as parameters, local variables, etc.

However, there is a separate *namespace* for types. Types can have the same name as fields, variables, or parameters, since they can be distinguished by their syntactic location in the program text. Also, methods and fields are in separate namespaces, as their use in the program text can be syntactically disambiguated. Hence, it is legal to have two symbols (members) in a class with the same name but one is a field and the other is a method.

According to the hiding rules above, local variables and parameters can shadow fields with the same name. Also, on the class scope, symbols of the super-classes get imported but possibly shadowed by definitions of symbols in the current class. Further, the order of definitions does not matter, *forward references* are allowed. These rules are the same as those that Java follows.

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.

Variables of a basic type are referred to by their identifier/name. Elements of an array are referred to with the `array[idx]` syntax, fields of a class instance are referred to using the standard `target.field` 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 field that is not defined. The compiler must flag such errors.

5 Methods

Methods can have an arbitrary number of parameters. For variables of both basic and reference type call-by-value is used. The evaluation of the parameters follows the "left to right" order. There is no overloading, and overriding methods must have the same return type as the overridden method.

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

Methods may return any of the previously defined legal types, or `void`.

5.1 Built-in methods

There are a few built-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 and for reference types (copying the reference).

7 Expressions

Expressions in Javali are evaluated like their Java equivalents [1].

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.

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. For a list of all errors, see Section B.

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.

Local variables, arrays, and class instances are zero-initialized after creation.

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

// reference to array of Structs
class Struct { int s; }
```

[1] In case of an expression where the operands may have side-effects the "left-hand operand first" or "left to right" rule applies.

```
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.

10.1 Runtime errors

Certain operations produce errors at runtime. In that case, the execution finishes with one of the following non-zero exit codes:

- **Invalid Downcast**: exit code 1, cast expression where runtime type is not a subtype of cast type
- **Invalid Array Store**: exit code 2, value of wrong type stored into covariant array (optional)
- **Invalid Array Bounds**: exit code 3, load or store into array with invalid index (optional)
- **Null pointer**: exit code 4, access field or invoke method on null pointer (optional)
- **Invalid Array Size**: exit code 5, create an array with a negative size (optional)
- **Possible Infinite Loop**: exit code 6, code runs for too long (interpreter only)
- **Division By Zero**: exit code 7, division by zero (interpreter only)
- **Internal Error**: exit code 22, some bug in the interpreter (interpreter only)

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 {
  ...
}
```

Specifying the extension of a class is optional. Subclasses inherit all methods and fields of their superclass(es). 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 runtime 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.

The `null` constant is compatible with all reference types.

When a reference `bref`, with 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 in the assignment will state if you can make this simplification.

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.

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.

```
// literals
Letter   ::=  "A".."Z" | "a".."z" .
Digit    ::=  "0".."9" .
HexDigit ::= Digit |  "a".."f"  |  "A".."F".
Integer  ::=  Decimal | Hex .
Decimal  ::=  "0" | "1".."9" { Digit }.
Hex      ::= ("0x" | "0X") HexDigit { HexDigit }
Boolean  ::= "true" | "false" .

Ident    ::=  Letter { Letter | Digit } .
Literal  ::= Integer | Boolean | "null" .

// types
PrimitiveType ::= "int" | "boolean" .
Type          ::= PrimitiveType | ReferenceType .
ReferenceType ::= Ident | ArrayType .
ArrayType     ::= Ident "[" "]"  |  PrimitiveType "[" "]" .

// program structure
Unit         ::= ClassDecl { ClassDecl } .
ClassDecl    ::= "class" Ident [ "extends" Ident ] "{" MemberList "}" .
MemberList   ::= { VarDecl | MethodDecl } .
VarDecl      ::= Type Ident { "," Ident } ";" .
MethodDecl   ::= ( Type | "void" ) Ident "(" [ FormalParamList ] ")"
                 "{" { VarDecl } { Stmt } "}" .
FormalParamList ::= Type Ident { "," Type Ident } .

// statements
Stmt      ::= AssignmentStmt | MethodCallStmt | IfStmt | WhileStmt
              | ReturnStmt | WriteStmt .
StmtBlock ::= "{" { Stmt } "}" .
MethodCallStmt ::= MethodCallExpr ";" .
AssignmentStmt ::= IdentAccess "=" ( Expr | NewExpr | ReadExpr ) ";" .
WriteStmt  ::= ( "write" "(" Expr ")" | "writeln" "(" ")" ) ";" .
IfStmt     ::= "if" "(" Expr ")" StmtBlock [ "else" StmtBlock ] .
WhileStmt  ::= "while" "(" Expr ")" StmtBlock .
ReturnStmt ::= "return" [ Expr ] ";" .
```

```
// expressions
NewExpr  ::= "new" ( Ident "(" ")" | Ident "[" Expr "]"
             | PrimitiveType "[" Expr "]" ) .
ReadExpr ::= "read" "(" ")".
MethodCallExpr ::= Ident "(" [ ActualParamList ] ")"
                   | IdentAccess "." Ident "(" [ ActualParamList ] ")" .
ActualParamList ::= Expr { "," Expr } .
IdentAccess ::= Ident | "this" | IdentAccess "." Ident
                | IdentAccess "[" Expr "]" | MethodCallExpr .
Expr ::= Literal | IdentAccess | "(" Expr ")"
         | ...see operators below... .
```

Table with operators ordered from highest precedence to lowest:

`("+" | "-" | "!") Expr`	Unary operators
`"(" ReferenceType ")" Expr`	Cast
`Expr ("*" | "/" | "%") Expr`	Multiplicative operators
`Expr ("+" | "-") Expr`	Additive operators
`Expr ("<" | "<=" | ">" | ">=") Expr`	Comparative operators
`Expr ("==" | "!=") Expr`	Equality operators
`Expr "&&" Expr`	Logical and
`Expr "||" Expr`	Logical or

Note: while not enforceable by the grammar, literals need to be in the range $[-2147483647, 2147483647]$ and otherwise produce a `ParseFailure`.

B Compile-time errors

You must guarantee that:

- There exists a class named `Main` with a method named `main` that takes no parameters, so that execution has a known starting point (`INVALID_START_POINT`).
- The superclass specified for each class exists. (`NO_SUCH_TYPE`)
- There is no circular inheritance (i.e., `A extends B` and `B extends A`). (`CIRCULAR_INHERITANCE`)
- No class is declared with the name `Object`. This name is reserved for a predefined class which shall serve as the root of the inheritance hierarchy. (`OBJECT_CLASS_DEFINED`)
- All classes have unique names. (`DOUBLE_DECLARATION`)
- There are no two fields with the same name in a single class. (`DOUBLE_DECLARATION`)
- There are no two methods with the same name in a single class. (`DOUBLE_DECLARATION`)

- Overridden methods must have the same number of parameters as the method from the superclass. (`INVALID_OVERRIDE`)
- The types of the parameters and the return type of overridden methods must be the same as for the corresponding method in the superclass. (`INVALID_OVERRIDE`)
- No method may have two parameters with the same name. (`DOUBLE_DECLARATION`)
- No method may have two local variables with the same name. (`DOUBLE_DECLARATION`)

Method bodies can contain the following kinds of errors:

- The built-in function `write()` should take a single integer argument. The number of arguments may be enforced by your parser, but the type of the expression must be checked by the semantic analyzer. (`TYPE_ERROR`)
- The built-in functions `read()`, and `writeln()` should take no arguments. In the framework, this is enforced by the parser and therefore does not cause a semantic failure.
  (`WRONG_NUMBER_OF_ARGUMENTS`).
- The condition for an if or while statement must be of boolean type. (`TYPE_ERROR`)
- For assignments, the type of the right-hand side must be a subtype of the type of the left-hand side. (`TYPE_ERROR`)
- Binary and unary arithmetic operators (`*, /, %, +, -`) must take arguments of integer type, and produce a result of integer. Binary arithmetic operators cannot be applied to values of two different types. (`TYPE_ERROR`)
- Binary and unary boolean operators (`!, &&, ||`) must take arguments of boolean type, and produce an boolean type result. (`TYPE_ERROR`)
- Relational operators (`<, <=, >, >=`) must take arguments of integer and produce an boolean type result. Relational operators cannot be applied to values of two different types. (`TYPE_ERROR`)
- Equality operators (`==, !=`) must take arguments of types `L` and `R` where either `L` is a subtype of `R`, or `R` is a subtype of `L`. (`TYPE_ERROR`)
- The built-in function `read()` produce an integer type.
- A cast from type `R` to type `C` is only legal if `R` is a subtype of `C` or vice-versa. (`TYPE_ERROR`)
- In a method invocation, the number of actual argument must be the same as the number of formal parameters in the method declaration. (`WRONG_NUMBER_OF_ARGUMENTS`)

- In a method invocation, the type of each actual argument must be a subtype of the type of the corresponding formal parameter. (`TYPE_ERROR`)
- All referenced fields must exist. (`NO_SUCH_FIELD`)
- All referenced methods must exist. (`NO_SUCH_METHOD`)
- The index to an array dereference (`x[i]`) must be of integer type, and the array must be of array type. The resulting type is the element type of the array. (`TYPE_ERROR`)
- When creating a new array, the length must be of integer type. (`TYPE_ERROR`)
- The constant `null` can be assigned to any reference type.
- The type name in a `new` statement or cast statement must be the name of a valid type. (`NO_SUCH_TYPE`)
- All referenced variables must be defined. (`NO_SUCH_VARIABLE`)
- The left-hand side of an assignment must not be a non-assignable expression (such as `this`). The only legal expressions are variables (`x`), fields (`x.f`), and array dereferences (`x[i]`). (`NOT_ASSIGNABLE`)
- There must be no attempt to access the field or method of a non-object type, such as an array or integer. (`TYPE_ERROR`)
- In a method return statement the expression type must be a subtype of the corresponding formal return type. (`TYPE_ERROR`)
- A method with a return type must have a return statement at the end of all its paths. (`MISSING_RETURN`)