VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

DATA STRUCTURES AND ALGORITHMS (CO2003)

Specification

JAVM - Just Another Virtual Machine

Version 1.1.0

1 Introduction

A virtual machine is a software program that can run on a physical machine to act as a virtual computer system with its own set of instructions and other components.

Virtual machines have many different applications:

- Simulator/emulator of the physical computer system on which softwares can be executed. (VirtualBox, VMWare, etc.)
- Platform to execute the same program in the same way on different physical systems. (Java virtual machine, etc.)
- Applications using a variety of complex data. (Microsoft Word with texts, images, tables, etc.)

The next sections will describe the components of JAVM (Just Another Virtual Machine), which is the object of interest in the assignments of this course.

2 About JAVM

The JAVM virtual machine is a stack-based machine (similar to Java Virtual Machine) that operates in a single-threaded mechanism. The main component of JAVM is a stack called JA Stack. Every time a function is invoked, a stack frame is created on the JA Stack and the entire function is executed in there.

Components of a stack frame include:

- Operand stack: a stack that used to push/pop operand values, results of operations, parameters passed to functions, return values of functions, etc.
- Local variable space: a space containing all of the local variables of the function being executed.

JAVM also has its own instruction set.

2.1 Data types

JAVM supports the data types in the following table:

Type	Value range	Size	Code
`boolean`	$0, 1$	1 byte	
`char`	$-2^7 \dots 2^7 - 1$	1 byte	
`short`	$-2^{15} \dots 2^{15} - 1$	2 bytes	
`int`	$-2^{31} \dots 2^{31} - 1$	4 bytes	0
`float`	32-bit IEEE 754 single-precision float	4 bytes	1

2.2 Operand stack

The operand stack is used to perform operations, store parameters to call a function, store return values of functions, etc. The operand stack has the same mechanism as a regular stack, only allowing data to be pushed onto/popped from the top of the stack. In addition to the data values, the stack also stores the data type of each element (the code of the data type) as an `integer` value. The maximum size of the stack is specified by the virtual machine.

The storge unit of JAVM's operand stack is word (4 bytes), so values of type `boolean`, `char` and `short` are cast to `int` when being pushed onto the stack.

The executing process of an operation on the operand stack is described in the following example.

Consider a math expression: $1 + 2 * 3 - 4.0$ described by the following instructions:

Instruction	Description	Operand stack
	Initial state	`<>`
`iconst 1`	Push 1 and 0 (code of `int` type) onto the stack.	`<1, 0>`
`iconst 2`	Push 2 and 0 (code of `int` type) onto the stack.	`<1, 0, 2, 0>`
`iconst 3`	Push 3 and 0 (code of `int` type) onto the stack.	`<1, 0, 2, 0, 3, 0>`
`imul`	Pop 2 elements from the stack's top Check types of the 2 elements (both are 0, valid) Perform multiplication, push the result 6 and the type code 0 onto the stack	`<1, 0, 6, 0>`
`iadd`	Pop 2 elements from the stack's top Check types of the 2 elements (both are 0, valid) Perform addition, push the result 6 and the type code 0 onto the stack	`<7, 0>`
`fconst 4.0`	Push 4.0 and 1 (code of `float` type) onto the stack	`<7, 0, 4.0, 1>`
`fsub`	Pop 2 elements from the stack's top Check types of the 2 elements (`int` and `float`, valid) Perform substraction, push the result 3.0 and the type code 1 onto the stack	`<3.0, 1>`

Thus, the top of the stack is the result of the expression and its data type's code.

2.3 Local variable space

The local variable space of each stack frame in JAVM is implemented using an AVL tree. The maximum size of the tree is specified by the virtual machine.

The location of an inserted node is defined by the comparison between two keys (two strings in this case). String a is considered less than string b when the value of the first character in a that does not match b is lower than that in b, or all compared characters in a match but the string a is shorter. Due to the property of the local variable space, that two nodes with the same key in the AVL tree at the same time is illegal.

The storage unit of JAVM's AVL tree is word (4 bytes), so values of type `boolean`, `char` and `short` are cast to `int` when being pushed into the tree. Each element includes the corresponding data type code (the type code is an `integer`) and the value of the variable being stored.

Consider these instructions: `int a = 1; int b = a;` described by the following instructions:

Instruction	Description	Local variable array	Operand stack
	Initial state	`[]`	`<>`
`iconst 1`	Push 1 and 0 (code of `int` type) onto the stack.	`[]`	`<1, 0>`
`istore a`	Pop the top element from the stack Check type of the element (`int` type, valid) Store the type code 0 and the value 1 into the tree with key `a`.	`{a: [0, 1]}`	`<>`
`iload a`	Get the value and type code of the variable `a` in the local variable space and push them onto the stack.	`{a: [0, 1]}`	`1, 0`
`istore b`	Pop the top element from the stack Check type of the element (`int` type, valid) Store the type code 0 and the value 1 into the tree with key `b`.	`{a: [0, 1], b: [0, 1]}`	`<>`

Therefore, the state of local variables at a certain time is stored in the local variable space.

2.4 Instructions

The instruction set of the JAVM virtual machine is divided into many groups. We need to pay attention to some main groups of instructions described in the following table:

No	Syntax	Description
Arithmetic Instructions		
1	`iadd`	Pop 2 element from the stack's top, check the data types' validity (`int`), perform the addition and push the result (type `int`) onto the stack. If the data type is not valid, the `TypeMismatch` exception will be thrown.
2	`fadd`	Similar to `iadd`, but the type is `float`.
3	`isub`	Pop 2 elements from the stack, check the data types' validity (`int`), perform subtraction (the top-most element is the second operand) and push the result (type `int`) onto the stack. If the data type is not valid, the `TypeMismatch` exception will be thrown.

No	Syntax	Description
4	`fsub`	Similar to `isub`, but the type is `float`.
5	`imul`	Pop 2 elements from the stack, check the data types' validity (`int`), perform multiplication and push the result (type `int`) onto the stack. If the data type is not valid, the `TypeMismatch` exception will be thrown.
6	`fmul`	Similar to `imul` but the type is `float`.
7	`idiv`	Pop 2 elements from the stack, check the data types' validity (`int`), perform the division (the top-most element is the second operand) and push the result (type `int`) onto the stack. If the data type is not valid, the `TypeMismatch` exception will be thrown. If the second operand is 0, the `DivideByZero` exception will be thrown.
8	`fdiv`	Similar to `idiv` but the type is `float`.
9	`irem`	Pop 2 elements from the stack, check the data types' validity (`int`), perform the modulo operation (the top-most element is the second operand) and push the result (type `int`) onto the stack. If the data type is not valid, the `TypeMismatch` exception will be thrown. If the second operand is 0, the `DivideByZero` exception will be thrown. Modulo operation `a rem b` is defined as `a - (a div b) * b`, where `div` is interger division operation.
10	`ineg`	Pop the first element from the stack, check the data types' validity (`int`), reverse the sign and push the result (type `int`) onto the stack. If the data type is not valid, the `TypeMismatch` exception will be thrown.
11	`fneg`	Similar to `ineg` but with type `float`.
12	`iand`	Pop 2 elements from the stack, check the data types' validity (`int`), perform the bitwise `"and"` and push the result (type `int`) onto the stack. If the data type is not valid, the `TypeMismatch` exception will be thrown.
13	`ior`	Pop 2 elements from the stack, check the data types' validity (`int`), perform the bitwise `"or"` and push the result (type `int`) onto the stack. If the data type is not valid, the `TypeMismatch` exception will be thrown.
14	`ieq`	Pop 2 elements from the stack, check the data types' validity (`int`), perform the equality comparison (the top-most element is the second operand) and push the result (0 if false, 1 if true, type `int`) onto the stack. If the data type is not valid, `TypeMismatch` exception will be thrown.
15	`feq`	Similar to `ieq` but the type is `float`. The result pushed onto stack has `int` type.
16	`ineq`	Pop 2 elements from the stack, check the data types' validity (`int`), perform the inequality comparison (the top-most element is the second operand) and push the result (0 if false, 1 if true, type `int`) onto the stack. If the data type is not valid, `TypeMismatch` exception will be thrown.
17	`fneq`	Similar to `ieq` but the type is `float`. The result pushed onto stack has `int` type.
18	`ilt`	Pop 2 elements from the stack, check the data types' validity (`int`), perform the `"less than"` comparison (the top-most element is the second operand) and push the result (0 if false, 1 if true, type `int`) onto the stack. If the data type is not valid, `TypeMismatch` exception will be thrown..
19	`flt`	Similar to `ieq` but the type is `float`. The result pushed onto stack has `int` type.

No	Syntax	Description
20	`igt`	Pop 2 elements from the stack, check the data types' validity (`int`), perform the `"greater than"` comparison (the top-most element is the second operand) and push the result (0 if false, 1 if true, type `int`) onto the stack. If the data type is not valid, the `TypeMismatch` exception will be thrown.
21	`fgt`	Similar to `ieq` but the type is `float`. The result pushed onto stack has `int` type.
22	`ibnot`	Pop the first element of the stack, check the data types' validity (`int`) and push the result (0 if the element's value is not equal to 0, or 1 if the element's value is equal to 0, the result's type is `int`) onto the stack. If the data type is not valid, the `TypeMismatch` exception will be thrown.
Load and Store Instructions		
23	`iconst <val>`	Push the value `<val>` (`int` type) onto the stack. `<val>` is an integer constant, which is made of digits 0-9 and a unary negative sign (-) if the value is negative.
24	`fconst <val>`	Similar to `iconst` with `float` type. `<val>` is a float constant, which is made of digits 0-9, a dot (.) among the numbers and a unary negative sign (-) if the value is negative.
25	`iload <var>`	Copy the value stored in the variable `var` from the local variable space, check the data types' validity (`int` type), and push the value onto the stack. If the data type is not valid, the `TypeMismatch` exception will be thrown. `<var>` is a string consisting of `a-zA-Z`.
26	`fload <var>`	Similar to `iload` but the type is `float`.
27	`istore <var>`	Pop the first element of the stack, check the data types' validity (`int` type) and save to the local variable space with key `var` (of type `int`). If the data type is not valid, the `TypeMismatch` exception will be thrown. `<var>` is a string consisting of `a-zA-Z`.
28	`fstore <var>`	Similar to `istore` but the type is `float`.
Type conversion Instructions		
29	`i2f`	Pop the top element from the stack, check the data types' validity (`int` type), cast the type to `float` and push the result (of type `float`) onto the stack. If the data type is not valid, the `TypeMismatch` exception will be thrown.
30	`f2i`	Pop the top element from the stack, check the data types' validity (`float` type), cast the type to `int` and push the result (of type `int`) onto the stack. If the data type is not valid, the `TypeMismatch` exception will be thrown.
Operand Stack Management Instructions		
31	`top`	Print to the console the top element of the stack (print the value, not the type code) together with a newline `\n` character. This instruction will not change the stack's state.
Local Variable Management Instructions		
32	`val <var>`	Print to the console the value of the variable `var` in the local variable space (print the value, not the type's code) together with a newline `\n` character. This instruction will not change the tree's state. `<var>` is a string consisting of `a-zA-Z`.

No	Syntax	Description
33	`par <var>`	Print to the console the name of the parent node variable of the variable `var` in the local variable space (print `"null"` instead if `var` is the root node) together with a newline `\n` character. This instruction will not change the tree's state. `<var>` is a string consisting of `a-zA-Z`.

Note: For instructions 1-22 that require operands of type `float`, if the type of an operand is `int`, the value will be cast to `float` type before performing the operation. In this case, no exception will be thrown.

2.5 Exceptions

The execution errors (exceptions) that need to be considered are described in the following table:

No	Exception	Description
1	`TypeMismatch(line)`	The exception occurs when the operands/elements taken from the operand stack or the local variable space do not match the type of the operation being executed.
2	`DivideByZero(line)`	The exception occurs when the second operand in a division operation is 0 or 0.0.
3	`StackFull(line)`	The exception occurs when attempting to push an element to the full operand stack.
4	`StackEmpty(line)`	The exception occurs when attempting to remove element from the empty operand stack.
5	`LocalSpaceFull(line)`	The exception occurs when storing data into the full local variable space.
6	`UndefinedVariable(line)`	The exception occurs when loading data from a memory area in the array of local memory space that has not been stored before.

If an instruction can cause various exceptions, the priority of exceptions is determined by the execution order of the instruction.

When throwing exceptions, the value `line`, which is the command line number that generates the error, should be included. The starting line of the program is 1. Right after throwing any exception, the programm is terminated and following instructions will not be executed.

Assume that no other exceptions other than the ones described above would occur.

3 Changelog

Version 1.0.1:

- Update descriptions about `irem`, `iload`, `istore`, `i2f`, `f2i`.
- Update descriptions about exception priority.

Version 1.1.0:

- Update descriptions about `f2i`.
- Change all descriptions to fit AVL tree local variable space.