Appendix A: The JCoCo Virtual Machine Specification

K.D. Lee, Foundations of Programming Languages, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-70790-7_9

JCoCo is a virtual machine which includes a built-in assembler. JCoCo executes assembly language programs by first processing the assembly language program and then executing it. The processing of the assembly language program is called *assembling*. The assembly language supported by JCoCo is defined by a BNF grammar. The grammar specifies how JCoCo assembly language programs are constructed. The grammar for the JCoCo virtual machine assembly language is provided in Fig. 9.1.

According to the BNF in Fig. 9.1 a JCoCo program is a sequence of class or function definitions. Each class definition may consist of one or more function definitions. Each function definition has several parts including a sequence of JCoCo virtual machine instructions like *LOAD_CONST*, *STORE_FAST*, and many others. The complete specification of instructions supported by JCoCo is provided at http://cs.luther.edu/~leekent/JCoCo and in this appendix. The complete syntax of the language is given in Fig. 9.1. There are just a few things to note in the BNF.

- Instructions may have as many labels defined on them as necessary. The definition of labeled instruction is recursive.
- The use of <null> indicates an empty production. For instance, a FunctionList may be empty meaning that there might not be a function list in a function definition. In this case that simply means a function might or might not have some nested functions.
- [and] indicate an optional part of a JCoCo program.
- Of course, the ... indicates there are more Unary and Binary mnemonics that are not listed in the BNF. The complete list of instructions and descriptions of each of them are given in this appendix.
- The JCoCo language is not line oriented. This BNF completely describes the language which has no line requirements. However, formatting a program like the disassembler will help in the clarity of written programs.

```
CoCoAssemblyProg ::= ClassFunctionListPart EOF

ClassFunctionListPart ::= ClassFunDef ClassFunctionList

ClassFunctionList ::= ClassFunDef ClassFunctionList | <null>

ClassFunDef ::= ClassDef | FunDef

ClassDef ::= Class colon Identifier [ ( Identifier ) ] BEGIN ClassFunctionList END

FunDef ::= Function colon Identifier slash Integer ClassFunctionList ConstPart
           LocalsPart FreeVarsPart CellVarsPart GlobalsPart BodyPart

ConstPart ::= <null> | Constants colon ValueList

ValueList ::= Value ValueRest

ValueRest ::= comma ValueList | <null>

Value ::= None | True | False | Integer |
          Float | String | code(Identifier) | TupleVal

TupleVal ::= ( ValueList )
(* the Scanner sees None, True, False, as Identifiers. *)

LocalsPart ::= <null> | Locals colon IdList

FeeVarsPart ::= <null> | FreeVars colon IdList

CellVarsPart ::= <null> | CellVars colon IdList

IdList ::= Identifier IdRest

IdRest ::= comma IdList | <null>

GlobalsPart ::= <null> | Globals colon IdList

BodyPart ::= BEGIN InstructionList END

InstructionList ::= <null> | LabeledInstruction InstructionList

LabeledInstruction ::= Identifier colon LabeledInstruction |
                       Instruction | OpInstruction

Instruction ::= STOP_CODE | NOP | POP_TOP | ROT_TWO | ROT_THREE | ...

OpInstruction ::= OpMnemonic Integer | OpMnemonic Identifier

OpMnemonic ::= LOAD_CONST | STORE_FAST | SETUP_LOOP | COMPARE_OP |
               POP_JUMP_IF_FALSE | ...
```

Fig. 9.1 The BNF for the JCoCo assembly language

9.1 Types

JCoCo supports the types given in Fig. 9.2.

Type	Description
type	the type of all types, including itself
NoneType	the type of None; None is a special value in Python referring to nothing. In other words, it is the null pointer in the JCoCo machine.
bool	the type of boolean types; True and False are the two boolean values.
int	integer types implemented as a C++ int; Depending on the C++ compiler this could be a 32-bit or 64-bit integer.
float	the type for floating point numbers; has the same precision as a C++ double precision floating point number.
str	the type of all strings; implemented using the C++ string class.
str_iterator	the type for iterators over strings
function	the type of all user-defined functions
method	the type of all user-defined methods
built_in_function_or_method	the type of all built-in functions or methods
range	the type of range objects; these are lazily generated sequences of integers
range_iterator	the type of range iterator objects; objects of this type yield consecutive integers in their associated range.
Exception	the type of all exceptions
list	the type of list objects like the original Python list objects
list_iterator	the type of iterators over lists
funlist	the type of functional list objects; This is a new type not supported in Python with the properties of lists from functional languages that are constructed from a head and a tail; funlist values are immutable as opposed to the list type.
funlist_iterator	the type of iterators over funlists.
tuple	Tuples are like lists, but are immutable.
tuple_iterator	the type of iterators over tuples
dict	the type of dictionaries, i.e. maps
dict_keyiterator	the type of dictionary key iterators
code	type of code objects (i.e. functions)
cell	the type of a reference objects
super	the type of super class objects used in the presence of inheritance
<class>	the type of any instance of the programmer-defined class.

Fig. 9.2 JCoCo supported types

9.2 JCoCo Magic and Attr Methods

One of the powerful features of the Python language comes from methods being looked up on objects at run-time. This means that new types of objects can easily be added to the language because the virtual machine instructions presented in this appendix will polymorphically call the proper methods since lookup happens at run-time. In support of this, JCoCo, like Python, has what have traditionally been called magic methods. These methods typically begin and end with two underscores. Magic methods are used by instructions as needed. For instance, the __add__ magic method is used by the BINARY_ADD instruction.

JCoCo includes support for all the magic methods that are defined by Python. While support is there for the whole list, not all magic methods are implemented on each type of object. The magic methods that are supported are controlled by the type of the object. When a magic method is called, the magic method is first looked up on the type and if it is supported, the call is made. Otherwise, an IllegalOperationEx-

ception is raised. The use of magic methods is illustrated in the descriptions of the JCoCo instructions in this appendix.

The possible magic methods include the following: __cmp__, __eq__, __ne__, __lt__, __gt__, __le__, __ge__, __pos__, __neg__, __abs__, __invert__, __round__, __floor__, __ceil__, __trunc__, __add__, __sub__, __mul__, __floordiv__, __div__, __truediv__, __mod__, __divmod__, __pow__, __lshift__, __rshift__, __and__, __or__, __xor__, __radd__, __rsub__, __rmul__, __rfloordiv__, __rdiv__, __rtruediv__, __rmod__, __rdivmod__, __rpow__, __rlshift__, __rand__, __ror__, __rxor__, __iadd__, __isub__, __imul__, __ifloordiv__, __idiv__, __itruediv__, __imod__, __ipow__, __ilshift__, __iand__, __ior__, __ixor__, __int__, __long__, __float__, __bool__, __cmplex__, __oct__, __hex__, __index__, __coerce__, __str__, __list__, vfunlist__, __repr__, __unicode__, __formatv, __hash__, __nonzero__, __dir__, __sizeofv, __getattr__, __setattr__, __delattr__, __getattribute__, __len__, __getitem__, __setitem__, __delitem__, __reversed__, __contains__, __missing__, __instancecheck__, __subclasscheckv, __call__, __copy__, __deepcopyv, __iter__, __next__, __type__, __excmatchv.

The last two magic methods are specific to CoCo and JCoCo but not a part of Python. The __type__ magic method is called when the type function is called on an object. The __excmatch__ magic method is called when matching an exception in an exception handler.

In addition, some objects have additional methods defined on them that are accessed like traditional method calls on objects. For instance, str objects have a split method that can be called to split a string on separator characters. The list of attr methods defined in JCoCo are *split* on strings, *append* on lists, *head* on funlists, *tail* on funlists, *concat* on strings, *keys* on dictionaries, and *values* on dictionaries. The *head* and *tail* methods are not found in Python but are defined in CoCo and JCoCo to support *funlist* objects which are defined to have a head and a tail.

9.3 Global Built-In Functions

JCoCo supports the following globally available built-in functions. These functions are not associated with any one type. When they are called, they polymorphically handle the arguments passed to them in their own manner as described.

print is a built-in function that prints a variable number of arguments to standard output, followed by a newline character, and returns None, just as print does in Python. The objects passed to print are printed by calling the __str__ magic method on each of them and appending their strings with an extra space between each pair of objects.

fprint prints exactly one argument. This is a built-in function that is specific to JCoCo and is not part of the standard Python language. It prints its argument by calling the __str__ magic method on the object to convert it to a string. This function returns itself, which can be useful when chaining together fprint expressions.

tprint prints exactly one argument, which may be a tuple, and returns None. tprint can be thought of as tuple print, because if a tuple is provided, the contents of the tuple are printed, separated by spaces, just as print does. However, tprint takes only one argument which may be a tuple. print takes a variable number of arguments. tprint is specific to JCoCo and is not part of the standard Python language. The values of the tuple are converted to strings using the __str__ magic method on each object. None is returned by tprint.

input is a built-in function that prints its prompt to standard output and returns one line of input as a string, just as input does in Python.

iter is a built-in function that constructs and returns an iterator over the object that is passed to it, just as Python's iter function works. This is implemented by calling the viter__ magic method on the object.

len is a built-in function that returns the length of the sequence that is passed to it. It does this by calling the __len__ magic method on the object given to it.

concat is built-in function that returns a string representation of the elements of its sequence concatenated together. The concat function in turn calls the concat method on the object that is passed to it.

int, **float**, **str**, **funlist**, **type**, and **bool** are all calls to types. When the type is called, the corresponding magic method of __int__, __float__, __strv, __funlistv, __type__, or __bool__ is called on the object that is passed to it. In this way, the object itself is in charge of how it is converted to the specified type.

range is a call to the range type that constructs a range object over the specified range. As in Python, the range function has 1, 2, or 3 arguments passed to it, representing the start, stop, and increment of the range of integer values. The start and increment values are optional.

Exception is a call to the exception type that constructs and returns an exception object that may be raised or thrown and caught by an exception handler.

super may be called in an instance method to gain access to the base class of an object. Single inheritance is supported in JCoCo. Unlike Python, multiple inheritance is not supported.

9.4 Virtual Machine Instructions

This is a subset of the full Python 3.2 instruction set with the addition of a few extra instructions and a couple of minor differences.

In the instructions in this appendix, TOS refers to the top element on the operand stack. TOS1 refers to the element second from the top of the operand stack. TOS2, and so on are similarly defined.

JCoCo instructions each take up exactly one location of space. The Python Virtual Machine uses one or more bytes for each instruction and therefore some instructions are composed of multiple bytes. JCoCo does not store its instructions as bytes and therefore each instruction takes exactly one location within the JCoCo virtual machine interpreter.

The Python Virtual machine defines some branching instructions as absolute jumps and other as relative jumps, that being relative to the current PC. JCoCo differs from the Python Virtual Machine in this regard. In the instructions any jump or branch is to an absolute location. Generally, the target of a branch or jump will

9.5 Arithmetic Instructions

BINARY_ADD

Implements TOS = TOS1 + TOS by making the call TOS1.__add__(TOS).

BINARY_SUBTRACT

Implements TOS = TOS1 - TOS by making the call TOS1.__sub__(TOS).

BINARY_MULTIPLY

Implements TOS = TOS1 * TOS by making the call TOS1.__mul__(TOS).

BINARY_MODULO

Implements TOS = TOS1 % TOS by making the call TOS1.__mod__(TOS).

BINARY_FLOOR_DIVIDE

Implements TOS = TOS1 // TOS by making the call TOS1.__floordiv__(TOS).

BINARY_TRUE_DIVIDE

Implements TOS = TOS1 / TOS by making the call TOS1.__truediv__(TOS).

BINARY_POWER

Implements TOS = TOS1 ** TOS by making the call TOS1.__pow__(TOS).

INPLACE_ADD

Implements in-place TOS = TOS1 + TOS. Exactly the same as BINARY_ADD by making the call TOS1.__add__(TOS).

9.6 Load and Store Instructions

BINARY_SUBSCR

Implements TOS=TOS1[TOS]. This instruction provides indexing into a list, tuple, or other object that supports subscripting. This is implemented as TOS1.__getitem__(TOS).

DELETE_FAST(namei)

This instruction does nothing in JCoCo which varies from the Python implementation. The purpose of this instruction seems to be implementation dependent. In the Python Virtual Machine it performs cleanup after an exception has

occurred. The handling of exceptions is different in JCoCo so this instruction exists to make it work with the disassembler, but it is ignored.

LOAD_ATTR(namei)

Replaces TOS with getattr(TOS,Globals[namei]). An attribute is usually a method associated with some object.

LOAD_CLOSURE(i)

Pushes a reference to the cell contained in slot i of the cell and free variable storage. The name of the variable is CellVars[i] if i is less than the length of CellVars. Otherwise it is FreeVars[i-len(CellVars)].

LOAD_CONST(consti)

Argument *consti* is a zero-based integer. Pushes Constants[consti] onto the stack.

LOAD_DEREF(i)

Loads the cell contained in slot i of the cell and free variable storage. Pushes a reference to the object the cell contains on the stack.

LOAD_FAST(namei)

Argument *namei* is a zero-based integer. Pushes a reference to Locals[namei] onto the stack.

LOAD_GLOBAL(namei)

Argument *namei* is a zero-based integer. Loads the Globals[namei] onto the stack.

LOAD_NAME(var_num)

Loads a value from the locals dictionary that is named in the globals at var_num. It pushes the loaded value onto the stack. Preference should be given to using LOAD_FAST if possible.

LOAD_BUILD_CLASS

Loads the built-in function for building classes onto the operand stack. This function, when called, takes two or three arguments. When there are two arguments passed to the build class function the name of the class must be at TOS. The function that will instantiate the class must be at TOS1. When called with CALL_FUNCTION, this built-in function for building classes will be at TOS2. This built-in function leaves the instantiated class on the top of the stack.

When called with three arguments the name of the base class, from which this class will inherit, is located at TOS and the other two arguments are in the same order under the name of the base class.

STORE_ATTR(var_num)

Stores the object found at TOS1 in the object found at TOS in an attribute name found in the globals at var_num.

STORE_DEREF(i)

Stores TOS into the cell contained in slot i of the cell and free variable storage.

STORE_FAST(namei)

Argument *namei* is a zero-based integer. Stores TOS into the Locals[namei].

STORE_LOCALS

Used during class instantiation. Pops the dictionary from TOS and uses it as the locals for the currently executing function, replacing any locals dictionary already in use. The popped dictionary is the attribute dictionary of the class which includes methods to be instantiated upon object instantiation for objects of the class.

STORE_NAME(var_num)

Uses the name found in the globals at var_num to store a named value in the locals dictionary. Preference should be given to STORE_FAST if possible.

STORE_SUBSCR

Implements TOS1[TOS]=TOS2. The instruction provides indexing into a mutable list or other object that supports subscripting. The instruction is implemented by calling TOS1.__setitem__(TOS,TOS2).

9.7 List, Tuple, and Dictionary Instructions

BUILD_MAP(initial_capacity)

Creates an empty dictionary object and pushes it onto the stack. The initial capacity is ignored by JCoCo.

STORE_MAP

Performs TOS2[TOS]=TOS1. TOS1 is the value to be stored at key TOS in dictionary TOS2.

BUILD_TUPLE(count)

Creates a tuple consuming count items from the stack, and pushes the resulting tuple onto the stack.

SELECT_TUPLE(count)

Pushes the contents of the tuple with count elements onto the operand stack. The count must match the tuple's size or an illegal operation exception will be thrown. The elements of the tuple are pushed so the left-most element is left on the top of the stack. This instruction is not part of the Python Virtual Machine. It is JCoCo specific.

BUILD_LIST(count)

Works as BUILD_TUPLE, but creates a list.

BUILD_FUNLIST

Works as BUILD_TUPLE, but creates a list.

SELECT_FUNLIST

This instruction pushes the head and the tail (which is a funlist) onto the operand stack. The head of the list is left on the top of the operand stack. The tail is below it on the stack. This instruction is JCoCo specific.

CONS_FUNLIST

Pops two elements from the operand stack. TOS should be a funlist and TOS-1 should be an element. The instruction create a new funlist from the two pieces with TOS-1 the head and TOS the tail of the new list. It pushes this new list onto the operand stack. This instruction is JCoCo specific.

9.8 Stack Manipulation Instructions

POP_TOP

Removes the top-of-stack (TOS) item.

ROT_TWO

Swaps the two top-most stack items.

DUP_TOP

Duplicates the reference on top of the stack.

9.9 Conditional and Iterative Execution Instructions

GET_ITER

Implements TOS=iter(TOS).

BREAK_LOOP

Terminates a loop due to a break statement.

POP_BLOCK

Removes one block from the block stack. Per frame, there is a stack of blocks, denoting nested loops, try statements, and such.

POP_EXCEPT

Removes one block from the block stack. The popped block must be an exception handler block, as implicitly created when entering an except handler. In

addition to popping extraneous values from the frame stack, the last three popped values are used to restore the exception state.

END_FINALLY

Terminates a finally clause. The interpreter recalls whether the exception has to be re-raised, or whether the function returns, and continues with the outer-next block.

COMPARE_OP(opname)

Performs a Boolean operation. Both TOS1 and TOS are popped from the stack and the boolean result is left on the operand stack after the execution of this instruction. opname is an integer corresponding to the following comparisons. In each case the comparison corresponding to opname is shown along with the magic method call that implements the comparison.

opname	Comparison Operation
0	TOS1 < TOS as TOS1.__lt__(TOS)
1	TOS1 <= TOS as TOS1.__le__(TOS)
2	TOS1 = TOS as TOS1.__eq__(TOS)
3	TOS1 != TOS as TOS1.__ne__(TOS)
4	TOS1 > TOS as TOS1.__gt__(TOS)
5	TOS1 >= TOS as TOS1.vge__(TOS)
6	TOS1 contains TOS as TOS1.__contains__(TOS)
7	TOS1 not in TOS as TOS1.__notin__(TOS)
8	TOS1 is TOS as TOS1.is_(TOS)
9	TOS1 is not TOS as TOS1.is_not(TOS)
10	Exception TOS1 matches TOS as TOS1.__excmatch__(TOS)

JUMP_FORWARD(target)

Sets the Program Counter (PC) to target.

POP_JUMP_IF_TRUE(target)

If TOS is true, sets the bytecode counter to target. TOS is popped.

POP_JUMP_IF_FALSE(target)

If TOS is false, sets the bytecode counter to target. TOS is popped.

JUMP_ABSOLUTE(target)

Set bytecode counter to target.

FOR_ITER(target)

TOS is an iterator. Call its __next__() method. If this yields a new value, push it on the stack (leaving the iterator below it). If the iterator indicates it is exhausted TOS is popped, and the PC is set to target.

SETUP_LOOP(target)

Pushes a block for a loop onto the block stack. The block spans from the current instruction to target.

SETUP_EXCEPT(target)

Pushes a try block from a try-except clause onto the block stack. Target points to the first except block.

SETUP_FINALLY(target)

Pushes a try block from a try-except clause onto the block stack. Target points to the finally block.

RAISE_VARARGS(argc)

This instruction varies from the Python version slightly. In JCoCo the argc must be one. This is because exceptions in JCoCo automatically contain the traceback which is not necessarily the case in the Python Virtual Machine. The argument on the stack should be an exception. The exception is thrown by this instruction.

9.10 Function Execution Instructions

RETURN_VALUE

Returns with TOS to the caller of the function.

CALL_FUNCTION(argc)

Calls a function. The *argc* indicates the number of positional parameters, the high byte the number of keyword parameters. On the stack, the opcode finds the keyword parameters first. For each keyword argument, the value is on top of the key. Below the keyword parameters, the positional parameters are on the stack, with the right-most parameter on top. Below the parameters, the function object to call is on the stack. Pops all function arguments, and the function itself off the stack, and pushes the return value.

MAKE_FUNCTION(argc)

Pushes a new function object on the stack. TOS is the code associated with the function. The function object is defined to have argc default parameters, which are found below TOS.

MAKE_CLOSURE(argc)

Creates a new function object, sets its closure, and pushes it on the stack. TOS is the code associated with the function, TOS1 the tuple containing cells for the closure's free variables. The function also has argc default parameters, which are found below the cells.

9.11 Special Instructions

BREAK_POINT

Pauses execution of the program and drops the JCoCo virtual machine into the interactive debugger.