Do compiler

Lassi Kortela
lassi@lassikortela.net

Ohjelmoinnin harjoitustyö
2010-04-22
Helsingin yliopisto
Avoin yliopisto
Kai Korpimies

Abstract

This document describes a Forth-derived programming language, Do, and a compiler that transforms it into Java source files and class files.

Table of contents

Do compiler user manual

Overview

The program is a compiler for the Do programming language. The language is described in more detail in the Overview section of the Do programming language specification included in this document.

The compiler currently outputs Java source files or equivalent binary class files for a Java Virtual Machine, henceforth JVM. It has been designed such that new backends for other output formats are easy to add.

Installation

The compiler runs on a JVM with read/write access to a file system. It is itself compiled by a Java compiler into Java class files that can be run by a JVM.

Usage

The compiler is invoked by one of the following command line commands.

Command	Input file name	Output file name	Output file format
do2java file	file.do	file.java	Java source file
do2class file	file.do	file.class	JVM class file

If the compiler encounters an error, it writes a message to the standard error stream and exits with status 1. If successful, it is silent and exits with status 0.

Due to time constraints during development, the error messages are extremely rudimentary: useful to a programmer familiar with the internals of the compiler, but fairly useless to others.

Do programming language specification

Overview

Do is a strictly and dynamically typed, garbage collected, stack-based, concatenative, imperative high-level programming language in the Forth language family.

Data types

none — Represents no object. Equivalent to the nil, null or none types in many languages.

boolean — Represents one of the two boolean truth values, true and false.

integer — Represents an integer, possibly negative.

string — Represents a string of Unicode characters.

vector — Represents a resizable vector containing a sequence of zero or more arbitrary objects.

The stack

One of the defining features of the Forth language family is the extensive use of stacks.

A stack is a collection of objects that supports insertion and removal of single objects. The insertion and removal operations are often called push and pop, respectively. The stack keeps track of the order in which the objects it contains were pushed. At any time only the last pushed object, termed the top of the stack, can be popped. This implies that items are popped in reverse order compared to the order in which they were pushed.

Do has a global stack that stores data objects that are manipulated by programs written in the programming language and is often used where local variables, function arguments or return values would be used in conventional languages. This stack is called "the stack".

Classic Forth also exposes another global stack, the return stack which tracks the function call hierarchy and enables user-visible continuations. Do currently doesn't expose its return stack.

The flag

A central invention in Do that is lacking from classic Forth is the flag, which is a global boolean variable that controls the behavior of conditional statements. The conditional statements in Do are & and | which return from the enclosing function if the flag has a particular value. We say that the flag is set when it has the value true and clear when it has the value false. Setting and clearing the flag mean assinging it that value.

Names

Like classic Forth, Do has a hyperstatic global environment. Hyperstatic global environment is the most hilariously baffling computer science term since metacircular interpreter. It simply means that the same name can be defined multiple times. Each reference to a name refers to the latest definition given before the reference. This is a good feature because often one can reuse a short name instead of being forced to invent unique names for similar operations. For example, if an algorithm has two nested loops, each can be called loop instead of one being loop1 and the other being loop2.

Definitions

A Do program is a sequence of definitions. There are two kinds of definitions: variable definitions and function definitions. Definitions cannot be nested.

Variable definitions

A variable definition consists of nothing but the name of the variable. It declares a global variable that is implicitly initialized to the value none at the beginning of the program. For a variable named var, the pseudo-functions var and var! are defined to get and set the value of the variable, respectively. The getter pushes the value onto the stack. The setter pops a value off the stack and assigns it to the variable.

Function definitions

A function definition consists of the name of the function followed by zero or more operations. When the function is called, the operations are carried out in sequence. The function returns to its caller when there are no more operations, or when a special primitive operation causes the function to return.

Operations

The following kinds of operations exist internally. All the calls and variable accesses look the same in source code; the behavior depends on the definition of the name in question.

- Call special primitive
- Call non-special primitive
- Call function
- Push literal integer
- Push literal string
- Get value of variable
- Set value of variable

Syntax

Tokens

Do has a simple character-level syntax: source code is a sequence of zero or more whitespace-delimited tokens. A token is a string, an integer, a symbol or a semicolon. Strings are delimited by double quotes. In addition, there are comments. Comments run from a
character until the end of the line. Semicolons stand by themselves. All other characters constitute bare tokens. If a bare token parses as an integer, it represents an integer. Otherwise it represents a symbol.

Some examlple tokens follow.

Token	Stands for

"foo"	the 3-letter string foo
"“"	a zero-length string
0	the number 0
10	the number 10
-10	the number -10
0xa	the number 10 in hex
foo	the 3-letter symbol foo
/mod	the 4-character symbol /mod
0=	the 2-character symbol 0=

Constructs

Token sequences are assembled into language constructs. There are only two kinds of language constructs recognized at the top level of nesting in the source file: let statements and function definitions.

A let statement begins with the symbol let. Zero or more symbols follow, each giving the name of a variable to be defined. The statement is terminated by a semicolon.

A function definition begins with any symbol besides let. Zero or more tokens follow, each giving an operation. An operation can be an integer, string or symbol token. An integer or string establishes itself as a literal that will be pushed to the stack by the operation. A symbol triggers one of the other operations listed in the Operations section.

Primitives

The following sections specify the primitive operations provided by a Do implementation. The format of a primitive specification is as follows.

name inputs — outputs ?

Written description

Where inputs and outputs are space-separated lists of zero or more objects on the stack. The inputs are expected to be on the stack when the primitive is called. The outputs will be on the stack when it returns. Objects whose names are in [brackets] may or may not be there; see the written description. A question mark after the outputs means that the primitive may alter the flag.

Special primitives

... —

Causes execution to continue at the beginning of the enclosing function.

& —

Returns from the enclosing function if the flag is clear.

| —

Returns from the enclosing function if the flag is set.

Data primitives

? obj — ?

Clears the flag if obj is none or false. Sets the flag otherwise.

not — ?

Toggles the flag.

drop obj —

Discards the top of the stack.

none — none

Pushes the object none.

true — true

Pushes the object true.

false — false

Pushes the object false.

Comparison primitives

= a b — a ?

Makes the flag indicate whether a is equal to b.

<> a b — a ?

Makes the flag indicate whether a is inequal to b.

<= a b — a ?

Makes the flag indicate whether a is less than or equal to b.

< a b — a ?

Makes the flag indicate whether a is less than b.

>= a b — a ?

Makes the flag indicate whether a is greater than or equal to b.

> a b — a ?

Makes the flag indicate whether a is greater than b.

Arithmetic primitives

+ a b — r

Pushes the sum of the numbers a and b.

- a b — r

Pushes the difference of the numbers a and b.

* a b — r

Pushes the product of the numbers a and b.

/mod a b — q r

Pushes the quotient and the remainder of a divided by b.

Sequence primitives

vector n — vec

Pushes a new vector whose initial items are **n** none-values.

count seq — n

Pushes the number of items in the sequence **seq**.

find subseq seq — [i] ?

If subseq occurs at leats once in seq, pushes the index of the first item of the first occurrence of subseq in seq and sets the flag. Otherwise clears the flag. Currently works on strings only.

push obj seq —

Inserts the object obj at the end of the sequence seq. Currently works on vectors only.

item i seq — i val

Pushes the **i**'th item of the sequence **seq**. If **seq** is a string, the result is a one-character substring.

item! i val seq — i

Replaces the **i**'th item of the sequence **seq** with the value **val**. Currently works on vectors only.

I/O primitives

readline — [str] ?

If standard input has reached its end, clears the flag. Otherwise reads the next line of input, pushes it as a string and sets the flag. The string does not contain a terminating newline, but any such newline is nevertheless consumed from the stream.

newline —

Writes a newline to and flushes standard output.

write obj —

Writes the object obj to and flushes standard output.

The written representation depends on the data type.

The values none, true and false are represented by those names.

An integer is represented in decimal notation.

A string is represented by itself.

A vector is represented by a left bracket, followed by the representations of the objects in it delimited by spaces, followed by a right bracket.

showstack —

Write a text representation of the current state of the stack to standard output.

die str —

Causes the program to write the given string to standard error and exit with status 1.

Example programs

Example Do programs are provided in the Do source code section in the appendix.

Do to Java translation notes

Limitations

Translation from Do to Java is not very efficient. The JVM internally uses an operand stack, but we can't really utilize that for the Do stack since it's not sharable across function call boundaries. It's apparently also not type safe. We resort to an ArrayList. As it stands, there is large method call overhead for simple things like pushing and popping the stack.

Name mangling

Identifiers in Do source code can contain almost any characters. In contrast, the range of characters allowed in Java identifiers is severely limited. Additionally, Do has a hyperstatic global environment whereas Java requires all global names to be unique. Consequently we have to make up a unique Java identifier for each Do identifier.

A Java identifier begins with an underscore, which is an easy way to make sure it doesn't conflict with any of the names defined in the Do runtime support code. The Do name is then appended with an underscore substituted for each non-alphanumeric character. Finally, an autoincrementing nonnegative integer is appended to make sure the name is unique.

A sample run of the name mangler follows.

Do name	Java name
foo	_foo0
foo	_foo1
/mod	__mod0
+mod	__mod1
+mod1	__mod10
+	__0

Data types

The Do data types map to Java data types as follows.

Do type	Java type
none	null
boolean	Boolean
integer	Integer
string	String
vector	ArrayList

Support file

The class files generated by the compiler do not stand alone. They need to be accompanied by a support file containing Java code that implements the Do runtime environment and non-special primitives. The following items appear in the support file.

The Do flag is a Java boolean variable.

The Do stack is a Java ArrayList.

Each non-special Do primitive is a hand-written Java function.

There are a few stack utilities used by the primitives: peek, drop, push, droppush, pop.

Program file

A single Do source file is translated into one public class within a single Java class file.

Each Do variable becomes a public static Java field of type Object.

Each Do function becomes a public static void Java method that throws Exception. The methods don't take any arguments, except for the main method, which takes the String[] argument that is mandated by Java.

As Java source code

The special Java primitives cause special control transfer code to be generated by the compiler. The body of each method representing a Do function is wrapped in an infinite for-loop. The & and | primitives are implemented as conditional Java break statements that test the Do flag and exit the infinite loop. The ... primitive is implemented as a Java continue statement that causes a control transfer to the beginning of the infinite loop. If no ... primitive appears in the function, a break statement is inserted as the last statement within the infinite loop.

As JVM bytecode

Pseudocode for JVM bytecode sequences representing the different kinds of Do operations follows. Check the source code and the instruction set section of the JVM specification for details.

...

```
goto funstart
```

&

```
getstatic DoPrim.flag Z
ifne x
return
x:
```

|

```
getstatic DoPrim.flag Z
ifeq x
return
x:
```

Call non-special primitive

```
invokestatic DoPrim.foo ()V
```

Call function

```
invokestatic DoProgram.foo ()V
```

Push literal integer

```
ldc_w idx
invokestatic DoPrim.push (I)V
```

Push literal string

```
ldc_w idx
invokestatic DoPrim.push (Ljava/lang/Object;)V
```

Get value of variable

```
getstatic DoProgram.varid Ljava/lang/Object;
invokestatic DoPrim.push (Ljava/lang/Object;)V
```

Set value of variable

```
invokestatic DoPrim.pop ()Ljava/lang/Object;
putstatic DoProgram.varid Ljava/lang/Object;
```

End of function

```
return
```

Do compiler implementation notes

Command classes

These classes implement the do2java and do2class commands, respectively, by chaining together a bunch of stream instances to create a pipeline that does the job. The classes consist of very straightforward glue code. The streams are chained in the following sequences.

To turn a Do source file into a do program definition:

- FileReader
- (PushbackReader)
- (CondReader)
- TokReader
- ConsReader
- ProgReader

To turn a Do program definition into a Java source file:

- JavaWriter
- FileWriter

To turn a Do program definition into a JVM class file:

- ClassWriter
- FileOutputStream

Compiler data classes

The data structures used by the compiler have been isolated into their own classes because they are used by multiple behavioral units, each of which is also in its own class. The data structure classes are tiny and very straightforward. It should be mentioned that some of them have methods for equality testing and writing out debugging information.

Classes in the Tok* hierarchy represent Do source code tokens.

Classes in the Cons* hierarchy represent Do source code constructs.

Classes in the Prog* hierarchy represent parts of a Do program definition.

The SyntaxError class represents a syntax error encountered by TokReader or ConsReader.

Compiler behavior classes

TokReader assembles Do source code tokens from characters supplied by a character stream. Each type of token has its own reading algorithm. The user of the class may also read tokens conditionally; for example, it may read the next token if and only if it is a symbol.

ConsReader assembles Do source code constructs from Do source code tokens supplied by a TokReader.

ProgReader assembles a Do program definition from Do source code constructs supplied by a ConsReader. It does name mangling and differentiates source constructs denoting operations within functions into their final forms.

JavaWriter writes out a Java source code file based on a Do program definition. This is a very straightforward undertaking.

ClassWriter writes out a JVM class file based on a Do program definition. The intricacy here arises mainly from the constant pool which needs to go at the beginning of the resulting class file but is referenced in parts of the file that follow it. This means that we have to write the parts that follow into a buffer in memory and assemble the actual file from the buffers when we're done. The code that writes out Java methods corresponding to Do functions is also intricate. Explaining it to people who don't have grounding in how the JVM works is outside the scope of this document.

Runtime support classes

The DoPrimID class is not actually part of the runtime environment, but is used by the compiler to translate the names of Do primitives into their Java IDs. The class is basically a simple table. See the discussion on name mangling for details.

The DoPrim class contains the runtime support common to all Do programs. See the discussion on Do to Java translation for details.

The DoPrimMIDlet class contains additional primitives used by Java ME programs on cell phones. It has never been tested because I didn't have enough time to wrestle with the mobile Java development tools.

Utility classes

The CmdLineProg class supports writing entry points to Java programs designed to be run as command line commands. The commmand will take one required argument, which is a file name. This is the source file name, and a corresponding destination file name is derived from it. The source and destination files must both have a particular file name extension determined by the caller. On the command line, the source file name extension may be omitted, in which case it is filled in.

The CondReader class is a character stream wrapper that conditionally reads characters if they match the given character class. A character class is represented as an integer predicate; see the explanation of the IntP class for details. Characters are actually ints, not chars, since we need to be able to represent the special values EOF and NOMATCH, which are negative so they don't conflict with char values.

The IntP class is a wrapper for an integer predicate — that is, a method that takes an integer argument and returns a boolean indicating whether that argument satisfies some condition. In the compiler, integer predicates are used to determine whether a particular character belongs to a particular character class; characters are represented as integers because we need to deal with pseudo-characters such as end-of-file. The IntP class has the static functions oneOf and notOneOf returning integer predicates that test whether a given integer is or is not part of the integers given to them as arguments. Each character class is encoded by a oneOf or notOneOf expression listing the characters that are included in or excluded from the class. These character classes drive the lexical analyzer.

The StrTab class is a pretty trivial string table. One gives it a string; it gives back the index of the string within the table. One can request the index of an old string that is already in the table, or a new string that is inserted into the table. One can also get the index of an old or new string; a new one is inserted only if there isn't an old one. The strings method returns the strings in the table as an array. In the compiler, StrTab is used to populate the string and primitive tables that are part of a program definition.

Testing report

The compiler was tested by writing the example programs and checking that they behave as expected when run. Additionally, a few intentionally invalid Do programs were written and it was checked that the compiler error-exited when told to compile them. With the finished compiler, all of the programs worked as expected.

One can test the compiler by the following method:

- Run javac *.java in the src/java directory
- Copy DoPrim.class into the src/do directory
- Open a Windows command prompt in the src/do directory
- For each .do program foo.do, type test foo to compile and run it

Limitations

JVM fixnums are not automatically promoted to bignums when their magnitude becomes too large to be represented as a fixnum. Instead, they silently wrap around, which is the same problem C has. The compiler doesn't insert code to work around the problem.

The error messages are lousy; the compiler basically says that a syntax error happened and leaves it at that. The Java backtraces are useful to compiler developers but not to ordinary users.

There is no no Do debugger, and no profiler. I didn't investigate the use of Java tools for this purpose.

Literal strings don't have an escape syntax.

The JVM class file backend emits tons of redundant constant pool entries because it doesn't keep a cache to remember which constants already have entries in the pool.

The JVM class file backend allocates all integer literals in the constant pool, even if they would fit into shorts and could therefore be inlined in the bytecode instruction stream. This bloats up the file and slows down execution.

The JVM class file backend doesn't try to emit opcode forms that are space-optimized. It usually uses the widest available form.

There are no compiler optimizations. I have a few in mind.

The Java compiler warns about dealing with ArrayLists that are not specialized to a subclass of Object, though such ArrayLists are useful and work perfectly well. It seems there is currently no way to fix the problem properly, but the cause is harmless and the resulting warnings can be ignored.

```
javac -Xlint:unchecked *.java
DoPrim.java:19: warning: [unchecked] unchecked call to add(E)
 as a member of the raw type java.util.ArrayList
        stack.add(x); }
```

Development diary

The development work was done in long stretches each spanning almost an entire day, with absolutely no development work done in between. Contrary to established time management wisdom, this is an extremely effective way to work.

Requirement gathering, design, implementation and testing were interspersed quite freely because the nature of non-trivial projects such as this implies that the phases cannot be completed in order. For example, one needs to write example programs to determine what primitives are needed in the programming language and whether they are natural to use. Then one needs to codify the primitives into a specification and implement the resulting programming language to make sure the example programs work and there have

been no oversights. Using the waterfall model for such a project is simply an inefficient use of time, if not beyond one's patience and mental capacity.

By the time the planning document was done, the parser was working and the language was solid enough that example programs had been written but not tested. As I write this document, the whole package is finished and works.

Changes from the planning document

A more thorough description of the Do programming language and the internals of the compiler were written. Information regarding translation of Do to Java and the JVM was added.