

Java compiler implementation in Python

Supervisor: Dr Jane Sinclair

Chris Lamb

March 2007

Motivation

Goal

To write an extendable and readable compiler in Python that targets the Java Virtual Machine (JVM).

Motivation

Goal

To write an extendable and readable compiler in Python that targets the Java Virtual Machine (JVM).

- Most compilers difficult to comprehend
- Sun's javac was not free software
- Investigate implementation issues
- Interest in compilers

Why Python?



Python is a multi-paradigm, high-level, strongly typed, dynamic and ‘duck’-typed programming language.

Why Python?



Python is a multi-paradigm, high-level, strongly typed, dynamic and ‘duck’-typed programming language.

- Rapid development – “show me the code” approach

Why Python?



Python is a multi-paradigm, high-level, strongly typed, dynamic and ‘duck’-typed programming language.

- Rapid development – “show me the code” approach
- Rich language

Why Python?



Python is a multi-paradigm, high-level, strongly typed, dynamic and ‘duck’-typed programming language.

- Rapid development – “show me the code” approach
- Rich language
- Good environment for test-driven development
(pyunit, doctest, no data hiding, etc.)

Why Python?



Python is a multi-paradigm, high-level, strongly typed, dynamic and ‘duck’-typed programming language.

- Rapid development – “show me the code” approach
- Rich language
- Good environment for test-driven development
(pyunit, doctest, no data hiding, etc.)
- Extensible – C extensions

Overview of the Java Virtual Machine

- Abstract computing machine
- Specified to interpret and execute Java bytecode

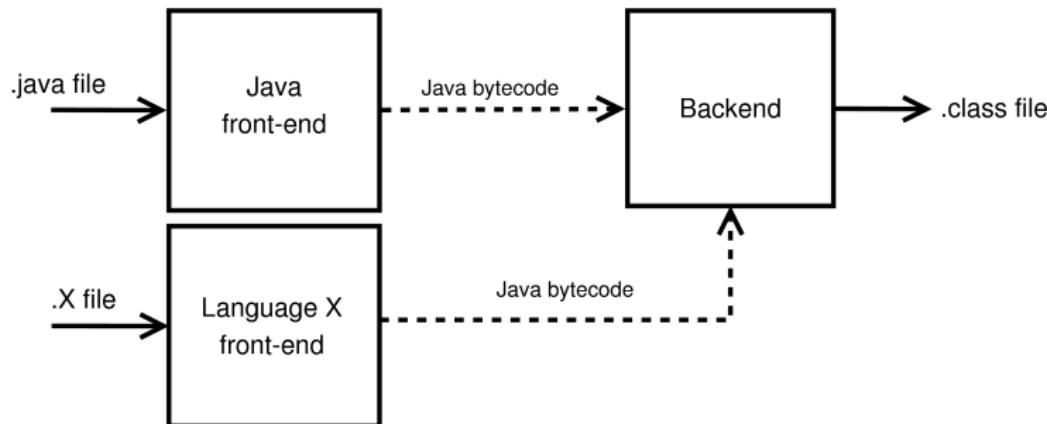
Overview of the Java Virtual Machine

- Abstract computing machine
- Specified to interpret and execute Java bytecode
- Stack-based
 - distinct operand stack for each method
 - computation performed on operand stack
 - method call stack

Overview of the Java Virtual Machine

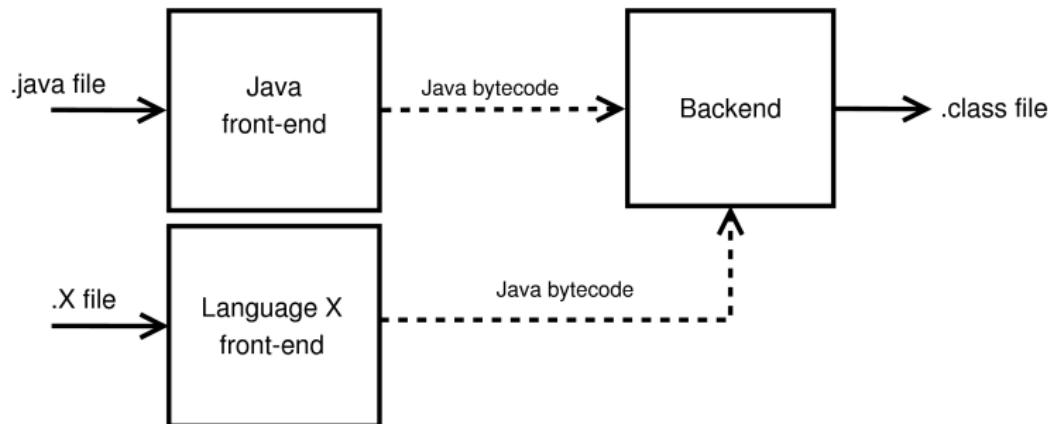
- Abstract computing machine
- Specified to interpret and execute Java bytecode
- Stack-based
 - distinct operand stack for each method
 - computation performed on operand stack
 - method call stack
- Local variables/registers
 - store intermediate results
 - implements Java local variables
 - used to accept arguments

Architecture overview



- Swappable front-ends

Architecture overview



- Swappable front-ends
- No intermediate code
 - Opcode language rich enough, and interferes when compiling other languages

Compiling Java

- Sequence of tasks/transformations
 - output from one task is the input to the next – ‘pipeline’
 - initial input is the .java file
 - final output is bytecode representation of input file
- Tasks perform transformations and tests
- Tasks implement Default Visitor Pattern [5]

Lexing and parsing

- Lexing splits input file into tokens
- Parsing validates tokens and constructs AST

Lexing and parsing

- Lexing splits input file into tokens
- Parsing validates tokens and constructs AST
- Java front-end uses 3rd-party parser generator, `python-ply`
- “Python Lex and Yacc” [3]
 - LALR(1) and SLR parsing methods
 - deliberately similar to *lex* and *yacc*
 - well-written
 - under active development

Type checking

Type checking

- Implicit casting → **short** s = 0; **int** i = s;
- Explicit casting → **int** i = 0; **short** s = (**int**) i;

Type checking

- Implicit casting → **short** s = 0; **int** i = s;
- Explicit casting → **int** i = 0; **short** s = (**int**) i;
- Reject statements that violate type safety
 - **if** ("This is a string") i = 4;

Type checking

- Implicit casting → **short** s = 0; **int** i = s;
- Explicit casting → **int** i = 0; **short** s = (**int**) i;
- Reject statements that violate type safety
 - **if** ("This is a string") i = 4;
- Detect uninitialised variables → **int** i; **int** j = i;

Type checking

- Implicit casting → **short** s = 0; **int** i = s;
- Explicit casting → **int** i = 0; **short** s = (**int**) i;
- Reject statements that violate type safety
 - **if** ("This is a string") i = 4;
- Detect uninitialised variables → **int** i; **int** j = i;
- Uses a type hierarchy, transitivity of types ($t_1.\text{type} = t_2.\text{type} + t_3.\text{type}$) and Aho's max and widen [1]
- Difficult to interpret the Java specification

Constant folding

- Pre-calculates known results at compile-time

Constant folding

- Pre-calculates known results at compile-time
- As an optimisation:
 - `int i = (1 + 2) * 3; ⇒ int i = 9;`

Constant folding

- Pre-calculates known results at compile-time
- As an optimisation:
 - `int i = (1 + 2) * 3; ⇒ int i = 9;`
- For detecting errors:
 - `byte b = 127 + 1; ⇒ loss of precision, print error`

Constant folding

- Pre-calculates known results at compile-time
- As an optimisation:
 - `int i = (1 + 2) * 3; ⇒ int i = 9;`
- For detecting errors:
 - `byte b = 127 + 1; ⇒ loss of precision, print error`
- For finding dead code:
 - primitive class variables marked `final` can have their value substituted
 - eg. `if (this.debug) System.out.println ("Debug statement");`

Linting

- Locates potential semantic errors in syntactically valid source code.

Linting

- Locates potential semantic errors in syntactically valid source code.
- Spot the error?
 - `for (int i = 0; i > 10; i++)`

Linting

- Locates potential semantic errors in syntactically valid source code.
- Spot the error?
 - `for (int i = 0; i > 10; i++)`
- Detecting no-ops:
 - `int i = 0; int j = (int) i;`

Register allocation

- Assigns Java variables to JVM local variables
- Goal is to minimise number of registers used

Register allocation

- Assigns Java variables to JVM local variables
- Goal is to minimise number of registers used

- Detects when variables are ‘live’
- Graph colouring algorithm ensures live variables are not overwritten

Register allocation

- Assigns Java variables to JVM local variables
- Goal is to minimise number of registers used
- Detects when variables are ‘live’
- Graph colouring algorithm ensures live variables are not overwritten
- More effective than Sun Microsystem’s allocator!
 - ie. **int** a, b; a = 1; b = 1 should only require 1 register
- Slow (graph colouring is NP-Hard)

Code generator

- Emits bytecode representation of AST

Code generator

- Emits bytecode representation of AST
- Very simple implementation
- Generates very naive code
 - we can optimise it later
 - simpler code \Leftrightarrow better maintainability

Code generator

- Emits bytecode representation of AST
- Very simple implementation
- Generates very naive code
 - we can optimise it later
 - simpler code \Leftrightarrow better maintainability

Optimisation philosophy

- The more optimisations we can do at the bytecode level, the more lazy front-end writers can be. :)

Other front-ends

Technique

Create isomorphisms between the semantics of the JVM and your programming language.

Other front-ends

Technique

Create isomorphisms between the semantics of the JVM and your programming language.

- Difficult
 - bytecodes statically typed
 - int-centric computation
 - single inheritance
 - single dispatch
 - overloading must follow Java's semantics

Other front-ends

Technique

Create isomorphisms between the semantics of the JVM and your programming language.

- Difficult
 - bytecodes statically typed
 - int-centric computation
 - single inheritance
 - single dispatch
 - overloading must follow Java's semantics
- Many workarounds, but “May Moore's law be with you.” [2]

Compiling BF

- Turing-complete language by Brian Raiter

```
stmt ::= '>' | '<' | '+' | '-' | '.' | '[' ( stmt )* ']'
```

Compiling BF

- Turing-complete language by Brian Raiter

```
stmt ::= '>' | '<' | '+' | '-' | '.' | '[' ( stmt )* ']'
```

```
int ptr = 0; char[] mem = new char[30000];
```

BF	Description	Java equivalent
>	Increment the pointer	ptr++;
<	Decrement the pointer	ptr--;
+	Increment the byte at the pointer	mem[ptr]++;
-	Decrement the byte at the pointer	mem[ptr]--;
.	Output the byte at the pointer	System.out.println (mem[ptr]);
stmt	Loop until byte at pointer is zero	while (mem[ptr] != 0) { stmt }

Compiling Lisp

```
(defun factorial (n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))

(print (factorial 8))
```

Compiling Lisp

```
(defun factorial (n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))

(print (factorial 8))
```

- Lisp program \Leftrightarrow static `main` method in Factorial class

Compiling Lisp

```
(defun factorial (n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))

(print (factorial 8))
```

- Lisp program \Leftrightarrow static `main` method in Factorial class
- **defun** definitions \Leftrightarrow static methods in Factorial class

Compiling Lisp

```
(defun factorial (n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))

(print (factorial 8))
```

- Lisp program \Leftrightarrow static `main` method in Factorial class
- **defun** definitions \Leftrightarrow static methods in Factorial class
 - puts **defuns** in the same namespace!
 - type-checker verifies correct scoping and assigns unique name

Compiler back-end

- Receives opcodes from front-end
- Optimises bytecode
- Assembles .class file to disk

Compiler back-end

- Receives opcodes from front-end
- Optimises bytecode
- Assembles .class file to disk
- Abstracts:
 - constant pool management
 - resolving labels / backpatching
 - calculating stack depth

Peephole optimisation

- Technique for “replacing instruction sequences by a shorter faster sequence whenever possible” [4]
- Replacement must retain the semantics of the original

Example peephole optimisation

Bytecode to increment local variable 0

```
...
iload_0
iconst_1
iadd
istore_0
...
```

⇒ iinc [localvar=0 delta=1]

Example peephole optimisation

Bytecode to increment local variable 0

```
...
iload_0
iconst_1
iadd
istore_0
...
```

⇒ iinc [localvar=0 delta=1]

Simple pattern matching not safe – need to ensure code cannot jump into the pattern, etc.

Defining pattern to match in Python

```
class iinc_shorthand(peephole.Optimisation):
    pattern = [
        opcodes.Load,
        opcodes.Constant,
        opcodes.iadd,
        opcodes.Store,
    ]
```

- Build pattern into definition of class.
- Pattern to match specified as a list of opcode classes
- Opcode classes in class hierarchy, allowing for generic rules

Semantic check of matched pattern

```
def test(self, splice):
    return splice[0].localvar == splice[3].localvar and \
        type(splice[1].value) is int and \
            0 <= splice[1].value <= 255 and \
                len(splice[1].labels) == len(splice[2].labels) == \
                    len(splice[3].labels) == 0
```

Attach a method to test:

- Load and Store refer to the same local variable,
- Constant refers to loading of an integer value,
- Constant is in the valid range for iinc, and
- no code can jump into the pattern

Calculating replacement sequence

```
def replace(self, splice):
    return [
        opcodes.iinc(splice[0].localvar, splice[1].value).
            merge_labels(splice[0])
    ]
```

replace method computes new opcode sequence:

- creates new iinc opcode
- merges labels from Load opcode

Algorithm splices result from replace where pattern was originally found.

Peephole optimisation

- Even few rules result in better code than Sun's javac!
- Simpler than adding a mini-language
- More maintainable than tons of conditionals

Peephole optimisation

- Even few rules result in better code than Sun's javac!
- Simpler than adding a mini-language
- More maintainable than tons of conditionals

- Still too verbose
- Limitation to sequence-matching
 - requires re-parsing of opcodes into an intermediate code, see Soot framework [6]

Future work

- Finish implementing Java language specification
- Better dead code analysis and linting
- Nicer error messages
- Java-level optimisations – loop unrolling, common subexpression elimination, hoisting, etc.
- Operand stack optimisations – dup
- Parser-generator – parse more advanced grammars
- Compiling dynamic languages – `invokedynamic` bytecode

Bibliography I

 Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman.

Compilers: Principles, techniques, & tools.
Addison-Wesley, second edition, 2007.

 Gilad Bracha.

Dynamically typed languages on the java platform (ts-3886).
In *JavaOne Technical Sessions*. Sun Microsystems, Inc., 2006.

 David M. Beazley (dave@dabeaz.com).

PLY (Python Lex-Yacc) documentation version 2.2, 2006.

 Jack W. Davidson and David B. Whalley.

Quick compilers using peephole optimization.

Software - Practice and Experience, 19(1):79–97, 1989.

Bibliography II

-  Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley Professional, 1994.
-  R. Vallee-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan.
Soot - a java bytecode optimization framework.
In *CASCON '99, 1999*, 1999.