Metaprogramming

These slides borrow heavily from Ben Wood's Fall '15 slides.

SOLUTIONS

CS251 Programming Languages
Spring 2018, Lyn Turbak

Department of Computer Science
Wellesley College

How to implement a programming language

Interpretation

An **interpreter** written in the **implementation language** reads a program written in the **source language** and **evaluates** it.

Translation (a.k.a. compilation)

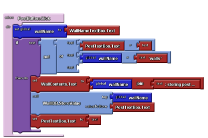
An **translator** (a.k.a. **compiler**) written in the **implementation language** reads a program written in the **source language** and **translates** it to an equivalent program in the **target language**.

But now we need implementations of:

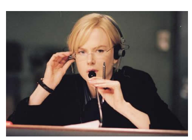
implementation language

target language

Metaprogramming: Interpretation



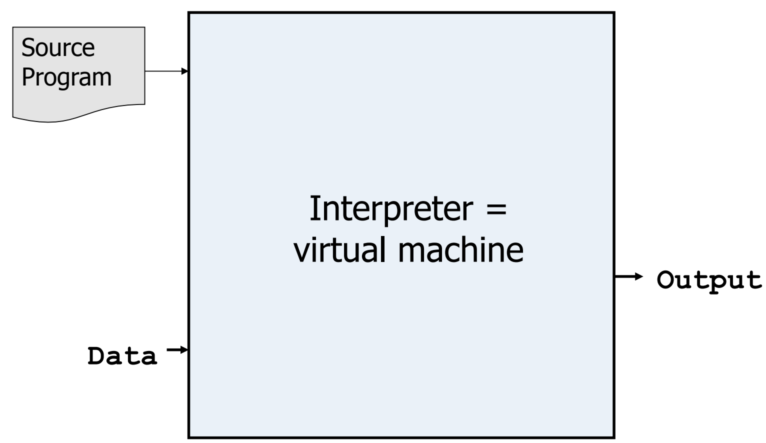
Program in language L



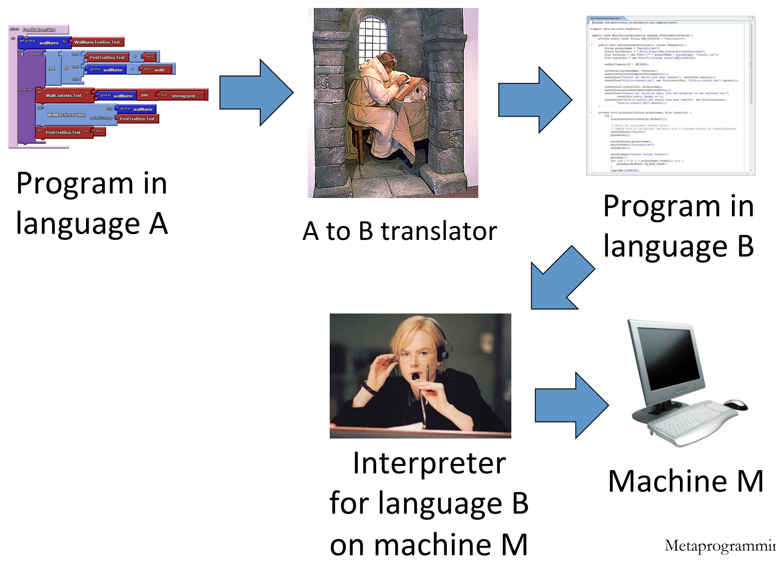
Interpreter for language L on machine M

Machine M

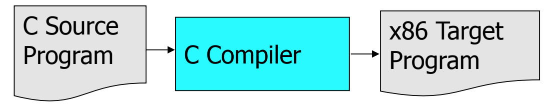
Interpreters



Metaprogramming: Translation

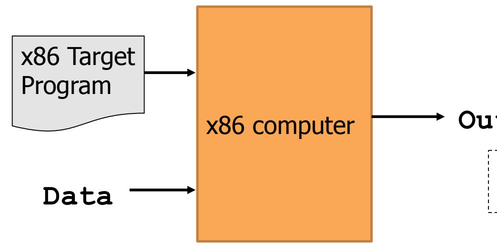


Compiler




```
if (x == 0) {
  x = x + 1;
}
...
```

```
cmp (1000), $0
bne L
add (1000), $1
L:
...
```



Thanks to **Ben Wood** for these and following pictures

Interpreters vs Compilers

Interpreters

No work ahead of time

Incremental

maybe inefficient

Compilers

All work ahead of time

See whole program (or more of program)

Time and resources for analysis and optimization

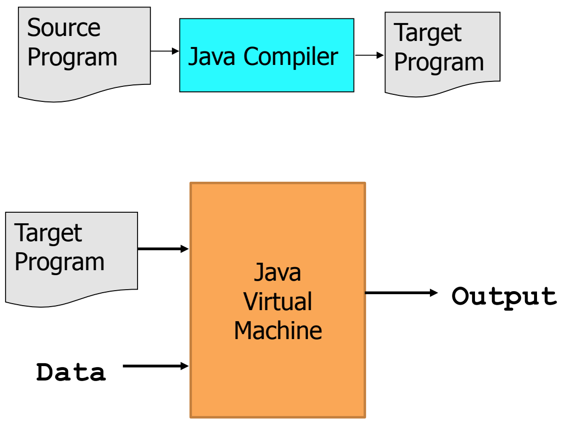
Java Compiler


```
if (x == 0) {
  x = x + 1;
}
...
```

```
load 0
ifne L
load 0
inc
store 0
L:
...
```

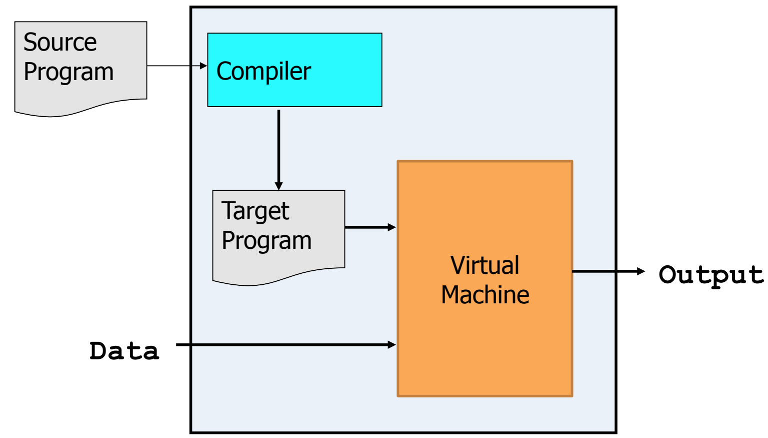
`(compare compiled C to compiled Java)`

Compilers... whose output is interpreted

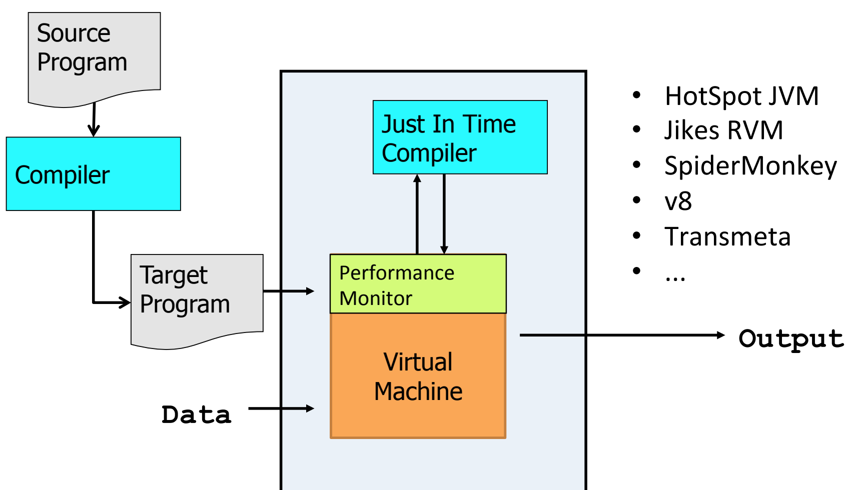


Doesn't this look familiar?

Interpreters... that use compilers.

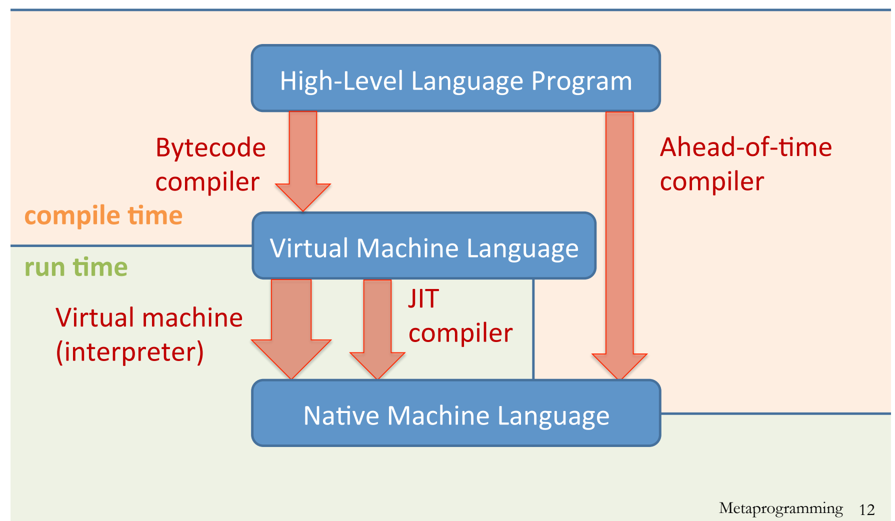


JIT Compilers and Optimization

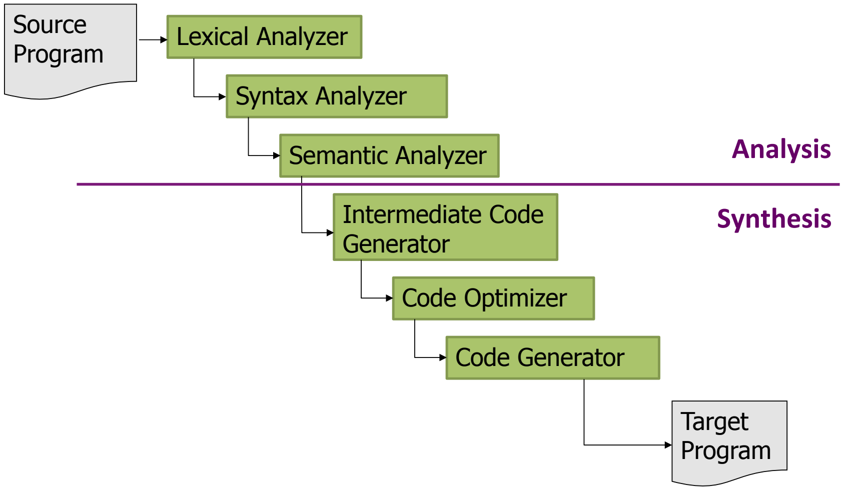


- HotSpot JVM
- Jikes RVM
- SpiderMonkey
- v8
- Transmeta
- ...

Virtual Machine Model



Typical Compiler



How to implement a programming language

Can describe by deriving a "proof" of the implementation using these inference rules:

Interpreter Rule

$$\frac{\textbf{P-in-L program} \qquad \textbf{L interpreter machine}}{\textbf{P machine}}$$

Translator Rule

$$\frac{\textbf{P-in-S program} \qquad \textbf{S-to-T translator machine}}{\textbf{P-in-T program}}$$

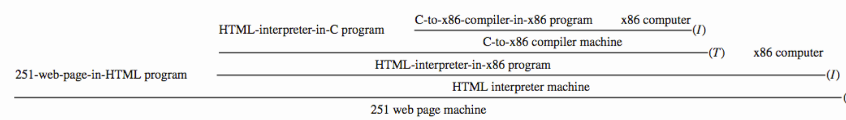
Implementation Derivation Example

Prove how to implement a "251 web page machine" using:

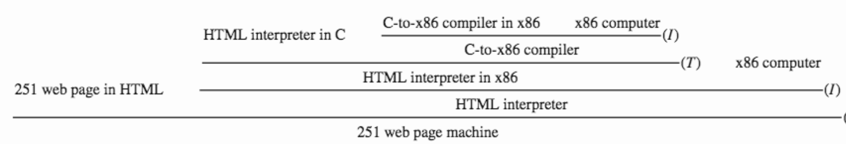
- 251-web-page-in-HTML program (a web page written in HTML)
- HTML-interpreter-in-C program (a web browser written in C)
- C-to-x86-translator-in-x86 program (a C compiler written in x86)
- x86 interpreter machine (an x86 computer)

No peaking ahead!

Implementation Derivation Example Solution



We can omit some occurrences of "program" and "machine":



Implementation Derivation Are Trees

And so we can represent them as nested structures, like nested bulleted lists:

- ❑ 251-web-page-in-HTML program
    - ○ HTML-interpreter-in-C program
        - • C-to-x86 compiler-in-x86 program
        - • X86 computer
    - ○ C-to-x86 compiler machine (I)
  - ✧ HTML-interpreter-in-x86 program (T)
  - ✧ x86 computer
- ❑ HTML interpreter machine (I)

251 web page machine (I)

Version that shows conclusions below bullets. More similar to derivations with horizontal lines, but harder to create and read

251 web page machine (I)

- ❑ 251-web-page-in-HTML program
- ❑ HTML interpreter machine (I)
  - ✧ HTML-interpreter-in-x86 program (T)
    - ○ HTML-interpreter-in-C program
    - ○ C-to-x86 compiler machine (I)
        - • C-to-x86 compiler-in-x86 program
        - • X86 computer
  - ✧ x86 computer

Preferred “top-down” version that shows conclusions above bullets.

Derivation Exercise

it's your turn

How to execute the Racket factorial program given these parts?

- ○ factorial-in-Racket program
- ○ Racket-to-Python-translator-in-Python program
- ○ Python-interpreter-in-C program
- ○ C-to-x86-translator-in-x86 program
- ○ x86 computer (i.e., x86 interpreter machine)

Warning: cannot start the following way:

factorial machine (I)
- ❑ factorial-in-Racket program
- ❑ Racket interpreter machine (I)
  ....

Why not?
The derivation would need to begin:

factorial machine (I)
- ❑ factorial-in-Racket program
- ❑ Racket interpreter machine (I)
  - ○ Racket-interpreter-in-L program
    ...
  - ○ L interpreter machine
    ...

But the parts don't include Racket-interpreter-in-L program **for any L!**

What to do? Explore translating the factorial-in-Racket program **to a** factorial-in-L program **for some L for which we *can* make an interpreter machine!**

Derivation Exercise: Solution

it's your turn

How to execute the Racket factorial program given these parts?

- ○ factorial-in-Racket program
- ○ Racket-to-Python-translator-in-Python program
- ○ Python-interpreter-in-C program
- ○ C-to-x86-translator-in-x86 program
- ○ x86 computer (i.e., x86 interpreter machine)

SOLUTION:

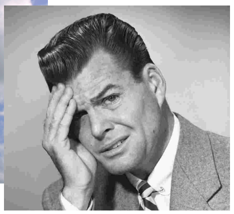
factorial machine (I)
- ❑ factorial-in-Python program (T)
  - ✧ factorial-in-Racket program
  - ✧ Racket-to-Python translation machine (I)
    - ➢ Racket-to-Python-translator-in-Python program
    - ➢ Python interpreter machine (I)
      - ◆ Python-interpreter-in-x86 program (T)
        - ○ Python-interpreter-in-C program
        - ○ C-to-x86-translator machine (I)
          - • C-to-x86-translator-in-x86 program
          - • x86 computer (= x86 interpreter machine
      - ◆ x86 computer (= x86 interpreter machine)
- ❑ Python interpreter machine (I)
  # Derivation already given above; no need to rederive it!
  # A reused derivation is a lemma, which corresponds to
  # a helper function in programming

Metaprogramming: Bootstrapping Puzzles

How can a Racket interpreter be written in Racket?

How can a Java compiler be written in Java?

How can gcc (a C-to-x86 compiler) be written in C?



Metacircularity and Bootstrapping

Many examples:

- Lisp in Lisp / Scheme in Scheme/Racket in Racket
- Python in Python: PyPy
- Java in Java: Jikes RVM, Maxine VM
- …
- C-to-x86 compiler in C: gcc
- `eval` construct in languages like Lisp, JavaScript

How can this be possible?

Key insights to bootstrapping:

- The first implementation of a language **cannot** be in itself, but must be in some other language.
- Once you have one implementation of a language L, you can can implement (enhanced versions of) L in L.

Metacircularity Example 1: Problem

Suppose you are given:

- Racket-interpreter-in-Python program
- Python machine
- Racket-interpreter-in-Racket program

How do you create a Racket interpreter machine using the Racket-interpreter-in-Racket program?

Metacircularity Example 1: Solution

Suppose you are given:

- Racket-interpreter-in-Python program
- Python machine
- Racket-interpreter-in-Racket program

How do you create a Racket interpreter machine using the Racket-interpreter-in-Racket program?

Racket interpreter machine #2 (I)
- ❑ Racket-interpreter-in-Racket program
- ❑ Racket-interpreter machine #1 (I)
  - ✧ Racket-interpreter-in-Python program
  - ✧ Python machine

But why create Racket interpreter machine #2 when you already have Racket-interpreter machine #1?

Metacircularity Example 1: More Realistic

Suppose you are given:

- Racket-subset-interpreter-in-Python program (implements only core Racket features; no desugaring or other frills)
- Python machine
- Full-Racket-interpreter-in-Racket-subset program

How do you create a Full-Racket interpreter machine using the Full-Racket-interpreter-in-Racket-subset program?

Full-Scheme interpreter machine (I)
- ❑ Full-Racket-interpreter-in-Racket-subset program
- ❑ Racket-subset interpreter machine #1 (I)
  - ✧ Racket-subset-interpreter-in-Python program
  - ✧ Python machine

Metacircularity Example 2: Problem

Suppose you are given:

- C-to-x86-translator-in-x86 program (a C compiler written in x86)
- x86 interpreter machine (an x86 computer)
- C-to-x86-translator-in-C program

How do you compile the C-to-x86-translator-in-C ?

Metacircularity Example 2: Solution

Suppose you are given:

- C-to-x86-translator-in-x86 program (a C compiler written in x86)
- x86 interpreter machine (an x86 computer)
- C-to-x86-translator-in-C program

How do you compile the C-to-x86-translator-in-C ?

C-to-x86-translator machine #2 (I)
- C-to-x86-translator-in-x86 program #2 (T)
    - C-to-x86-translator-in-C
    - C-to-x86-translator machine #1 (I)
        - C-to-x86-translator-in-x86 program #1
        - x86 computer
- x86 computer

But why create C-to-x86-translator-in-x86 program #2 (T)
when you already have C-to-x86-translator-in-x86 program #1?

Metacircularity Example 2: More Realistic

Suppose you are given:

- C-subset-to-x86-translator-in-x86 program
  (a compiler for a subset of C written in x86)
- x86 interpreter machine (an x86 computer)
- Full-C-to-x86-translator-in-C-subset program
  (a compiler for the full C language written in a subset of C)

How do you create a Full-C-to-x86-translator machine ?

Full-C-to-x86-translator machine (I)
- Full-C-to-x86-translator-in-x86 program (T)
    - Full-C-to-x86-translator-in-C-subset
    - C-subset-to-x86-translator machine (I)
        - C-subset-to-x86-translator-in-x86 program
        - x86 computer
- x86 computer

A long line of C compilers

C-version_n-to-target_n-translator machine (I)
- C-version_n-to-target_n-translator program in target_n-1 (T)
    - C-version_n-to-target_n-translator program in C-version_n-1
    - C-version_n-1-to-target_n-1 translator machine (I)
        - C-version_n-1-to-target_n_1-translator program in target_n-2 (T)

          ⋮

            - C-version_2-to-target_2-translator-program in target_1 (T)
                - C-version_2-to-target_2-translator program in C-version_1
                - C-version_1-to-target_1 translator machine (I)
                    - C-version_1-to-target_1-translator program in assembly_0
                    - assembly_0 computer
            - target_1 computer

          ⋮

        - target_n-2 computer
- target_n-1 computer

- The versions of C and target languages can change at each stage.
- Trojan horses from earlier source files can remain in translator machines even if they're not in later source file! See Ken Thompson's *Reflection on Trusting Trust*

More Metaprogramming in SML

- We've already seen PostFix and s-expressions in Racket; next we'll see how to implement these in SML
- The rest of the course explores a sequence of expression languages implemented in SML that look closer and closer to Racket:
  - Intex: a simple arithmetic expression language
  - Bindex: add naming to Intext
  - Valex: add more value types, dynamic type checking, desugaring to Bindex
  - HOFL: add first class function values, closure diagrams to Valex
  - HOFLEC: add explicit SML-like mutable cells to HOFL

Remember: language != implementation

- Easy to confuse "the way this language is usually implemented" or "the implementation I use" with "the language itself."
- Java and Racket can be compiled to x86
- C can be interpreted in Racket
- x86 can be compiled to JavaScript
- Can we compile C/C++ to Javascript? http://kripken.github.io/emscripten-site/