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

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

- Program in language A
- A to B translator
- Program in language B
- Interpreter for language B on machine M
- Machine M

Compiler

- C Source Program
- C Compiler
- x86 Target Program

\[
\text{if } (x == 0) \{ \\
\quad x = x + 1; \\
\}
\]

\[
\text{cmp (1000), } $0 \\
\text{bne L} \\
\text{add (1000), } $1 \\
\text{L: } \\
\]

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

- Source Program
- Java Compiler
- Target Program

\[
\text{if } (x == 0) \{ \\
\quad x = x + 1; \\
\}
\]

\[
\text{load 0} \\
\text{ifne L} \\
\text{load 0} \\
\quad \text{inc} \\
\text{store 0} \\
\text{L: } \\
\]

Thanks to Ben Wood for these and following pictures
**Compilers... whose output is interpreted**

Source Program → **Java Compiler** → Target Program

Target Program → **Java Virtual Machine** → Output

Doesn't this look familiar?

**Interpreters... that use compilers.**

Source Program → **Compiler** → Target Program

Target Program → **Virtual Machine** → Output

**Typical Compiler**

Source Program → **Lexical Analyzer** → **Syntax Analyzer** → **Semantic Analyzer** → **Intermediate Code Generator** → **Code Optimizer** → **Code Generator** → **Target Program**

Analysis

Synthesis

**JIT Compilers and Optimization**

Source Program → **Compiler** → **Performance Monitor** → **Just In Time Compiler** → **Target Program** → **Virtual Machine** → Output

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

**Metaprogramming**
Virtual Machine Model

How to implement a programming language

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

Interpreter Rule

<table>
<thead>
<tr>
<th>P-in-L program</th>
<th>L interpreter machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>P machine</td>
<td></td>
</tr>
</tbody>
</table>

Translator Rule

<table>
<thead>
<tr>
<th>P-in-S program</th>
<th>S-to-T translator machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-in-T program</td>
<td></td>
</tr>
</tbody>
</table>

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 peeking 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)

**Metaprogramming: Bootstrapping Puzzles**
How can we write Scheme interpreter in Scheme?
How can we write a Java-to-x86 compiler in Java?

**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
- 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, you can implement it in itself.

**Metacircularity Example 1: Problem**
Suppose you are given:
- Scheme-interpreter-in-Python program
- Python machine
- Scheme-interpreter-in-Scheme program

How do you create a Scheme interpreter machine using the Scheme-interpreter-in-Scheme program?
**Metacircularity Example 1: Solution**

Suppose you are given:
- Scheme-interpreter-in-Python program
- Python machine
- Scheme-interpreter-in-Scheme program

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

<table>
<thead>
<tr>
<th>Scheme interpreter machine #2 (I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheme-interpreter-in-Scheme program</td>
</tr>
<tr>
<td>Scheme-interpreter machine #1 (I)</td>
</tr>
<tr>
<td>Scheme-interpreter-in-Python program</td>
</tr>
<tr>
<td>Python machine</td>
</tr>
</tbody>
</table>

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

---

**Metacircularity Example 1: More Realistic**

Suppose you are given:
- Scheme-subset-interpreter-in-Python program (implements only core Scheme features; no desugaring or other frills)
- Python machine
- Full-Scheme-interpreter-in-Scheme program

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

<table>
<thead>
<tr>
<th>Full-Scheme interpreter machine (I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheme-interpreter-in-Scheme program</td>
</tr>
<tr>
<td>Scheme-subset interpreter machine #1 (I)</td>
</tr>
<tr>
<td>Scheme-subset-interpreter-in-Python program</td>
</tr>
<tr>
<td>Python machine</td>
</tr>
</tbody>
</table>

---

**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-subset 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?

<table>
<thead>
<tr>
<th>C-to-x86-translator machine #2 (I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-to-x86-translator-in-x86 program #2 (T)</td>
</tr>
<tr>
<td>C-to-x86-translator-in-C</td>
</tr>
<tr>
<td>C-to-x86-translator machine #1 (I)</td>
</tr>
<tr>
<td>C-to-x86-translator-in-x86 program #1</td>
</tr>
<tr>
<td>x86 computer</td>
</tr>
<tr>
<td>x86 computer</td>
</tr>
</tbody>
</table>

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

![Diagram showing the process]

More Metaprogramming in SML

- We’ve already seen PostFix and Intex SML
- A sequences of expression languages implemented in SML that look closer and closer to Racket:
  - Bindex: add naming
  - Valex: add more value types, dynamic type checking, desugaring
  - HOFL: first class function values, closure diagrams

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/emshepten-site/]