Metaprogramming

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

Metaprogramming: Interpretation

Interpreters

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: Translation

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

Interpreter

Metaprogramming

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

Compiler

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

\[...
\]

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

\[...
\]

x86 Target Program

x86 computer

Output

Thanks to Ben Wood for these and following pictures

Java Compiler

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

\[...
\]

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

\[...
\]

(compare compiled C to compiled Java)
Compilers... whose output is interpreted

- Source Program → Java Compiler → Target Program
  - Java Virtual Machine
  - Output

Doesn't this look familiar?

Interpreters... that use compilers.

- Source Program → Compiler
  - Target Program → Virtual Machine → Output
  - Data

JIT Compilers and Optimization

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

Virtual Machine Model

- High-Level Language Program → Bytecode compiler → Virtual Machine Language
  - JIT compiler
  - Ahead-of-time compiler
  - Native Machine Language
How to implement a programming language

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

**Interpreter Rule**

\[
\begin{align*}
P \rightarrow L \rightarrow P \rightarrow L \\
\text{P-in-L program} & \quad L \text{ interpreter machine} \\
\text{P machine} & \\
\end{align*}
\]

**Translator Rule**

\[
\begin{align*}
\text{P-in-S program} & \quad \text{S-to-T translator machine} \\
\text{P-in-T program} & \\
\end{align*}
\]

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 Are Trees

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

<table>
<thead>
<tr>
<th>251-web-page-in-HTML program</th>
</tr>
</thead>
<tbody>
<tr>
<td>o HTML-interpreter-in-C program</td>
</tr>
<tr>
<td>o C-to-x86 compiler-in-x86 program</td>
</tr>
<tr>
<td>• X86 computer</td>
</tr>
<tr>
<td>o C-to-x86 compiler machine (I)</td>
</tr>
<tr>
<td>o HTML-interpreter-in-x86 program (T)</td>
</tr>
<tr>
<td>• x86 computer</td>
</tr>
</tbody>
</table>

| 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.

Preferred "top-down" version that shows conclusions above bullets.

### Derivation Exercise

How to execute the Racket factorial program given these parts?

**Warning:** cannot start the following way:

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

**Why not?**

### Derivation Exercise: Solution

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)

Put your solution here:

### 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 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-Racket 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?

<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>o C-to-x86-translator-in-x86 program #1</td>
</tr>
<tr>
<td>o x86 computer</td>
</tr>
<tr>
<td>q 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?

### 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?

<table>
<thead>
<tr>
<th>Full-C-to-x86-translator machine (I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full-C-to-x86-translator-in-x86 program (T)</td>
</tr>
<tr>
<td>Full-C-to-x86-translator-in-C-subset</td>
</tr>
<tr>
<td>C-subset-to-x86-translator machine (I)</td>
</tr>
</tbody>
</table>
<pre><code>| o C-subset-to-x86-translator-in-x86 program |
| o x86 computer |
</code></pre>
<p>| q x86 computer |</p>

### A long line of C compilers

<table>
<thead>
<tr>
<th>C-version_n-to-target_n-translator machine (I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-version_n-to-target_n-translator program in target_n-1 (T)</td>
</tr>
<tr>
<td>C-version_n-to-target_n-translator program in C-version_n-1</td>
</tr>
<tr>
<td>C-version_n-1-to-target_n-1 translator machine (I)</td>
</tr>
</tbody>
</table>
<pre><code>| o 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 |
</code></pre>
<p>| q target_n-1 computer |</p>

- 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
Bootstrapping: Mary Allen Wilkes ‘59

Created LAP operating system for Wesley A. Clark’s LINC computer, widely regarded as the first personal computer (designed for interactive use in bio labs). Work done 1961—1965. Created first interactive keyboard-based text editor on 256 character display. LINC had only 2K 12-bit words; (parts of) editor code fit in 1K section; document in other 1K.

In 1965, she developed LAP6 with LINC in Baltimore living room.

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
  - HOILEC: 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/]