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

---

SOLUTIONS

CS251 Programming Languages
Spring 2018, Lyn Turbak
Department of Computer Science
Wellesley College

---

Metaprogramming

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

---

**Metaprogramming: Interpretation**

Program in language L $\rightarrow$ Interpreter for language L on machine M $\rightarrow$ Machine M

---

**Interpreters**

Interpreter = virtual machine

---
**Metaprogramming: Translation**

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

**Compiler**

```
if (x == 0) {
    cmp (1000), $0
    bne L
    add (1000), $1
    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**

```
if (x == 0) {
    load 0
    ifne L
    load 0
    inc
    store 0
    L:
}
```

Thanks to Ben Wood for these and following pictures.

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

Interpreters... that use compilers.

JIT Compilers and Optimization

Virtual Machine Model

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

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

Analysis

Synthesis

How to implement a programming language

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

Interpreter Rule

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

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 Example Solution

We can omit some occurrences of “program” and “machine”:

No peeking ahead!
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

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?
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!

SOLUTION:

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-**subset**-interpreter-in-Racket-subset program
  - Racket-**subset** interpreter machine #1 (I)
    - Racket-**subset**-interpreter-in-Python program
    - Python machine
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: Problem**

**Metaprogramming**

**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?

**A long line of C compilers**

Suppose you are given:

- C-subset-to-x86-translator-in-x86 program (a compiler for a subset of C written in x86)
- 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
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/]