Specifying Semantics
How Interpreters Work

CS 331 Programming Languages

Lecture Slides

Wednesday, April 2, 2025

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

Unit Overview
The Scheme Programming Language

Topics

- ✓ PL feature: identifiers & values
- ✓ PL feature: reflection
- ✓ PL category: Lisp-family PLs
- ✓ Introduction to Scheme
- ✓ Scheme: basics
- ✓ Scheme: evaluation
- ✓ Scheme: data
- ✓ Scheme: macros

DONE

Unit Overview
Semantics & Interpretation

Topics

✓ - Introduction to semantics
- Specifying semantics
- How interpreters work
- Writing an interpreter

Review

Review
Introduction to Semantics

Syntax is the correct *structure* of code.

Semantics is the *meaning* of code.

Some things the semantics of a PL might address:

Dynamic Semantics	Runtime behavior of code	"`a + b`" computes the sum of the values of `a`, `b` and returns the result.
Static Semantics (in some PLs)	Static typing	"`int n;`" declares a variable named `n` of type `int`.
	Data structure organization	"`class Zz { int _a, _b; };`" declares a class named `Zz` with two private data members: `_a`, `_b`.
	Correctness of cases	"`switch (n) { case 1: case 1: a=0; }`" is an illegal switch-statement, because it contains duplicate cases.

Specifying Semantics

Specifying Semantics
Introduction [1/3]

From the first day of class.

Consider. Alice invents a PL and writes a precise description of it—a **specification**. Now Bob and Carol want to write compilers for this PL.

With a properly written specification, Bob will be able to write a compiler without talking to Alice. Carol will be able to write a compiler without talking to Alice or Bob. The two compilers will compile the same programs. The executables produced by these compilers will do the same things.

How does Alice write a specification? How do Bob and Carol use it?

We have answered these questions as they relate to specifying the syntax of a PL. Now, what about the semantics of a PL?

Specifying Semantics
Introduction [2/3]

A **formal specification method** is a mathematically based technique for describing something. Formal specifications use precisely defined notation. Other methods are **informal**.

For example, we can formally specify a language using a regular expression: */xy*/*

Or we can informally specify it by describing it in words: strings that consist of an *x* character followed by zero or more *y* characters.

We have looked at formal methods for specifying the *syntax* of a PL—in particular, phrase-structure grammars.

Formal semantics refers to formal specification methods for semantics.

Specifying Semantics
Introduction [3/3]

We look very briefly at two formal-semantics specification methods. *We will not cover notation.*

- **Operational semantics**. Specify semantics of a PL in terms of the semantics of some other PL or abstract machine (usually the latter).
- **Denotational semantics**. Specify semantics by representing state & values with mathematical objects, commands & computations by functions.

Operational semantics is the name given to a number of methods for specifying semantics. These focus on the actions or computations that various pieces of code perform.

There are a number of different kinds of operational semantics. What they have in common is that the actions/computations of code are expressed in terms of some other system whose semantics is already known. This system might be:

- A mathematical system.
- An abstract machine—perhaps some kinds of automaton.
- Some other programming language.

For operational semantics to be worth using, this other system must be precisely specified, and also simpler than the PL whose semantics is being described.

Specifying Semantics
Denotational Semantics [1/2]

Denotational semantics is another method for specifying semantics. It involves the construction of mathematical objects called **denotations**, which describe the meanings of program entities. The denotation of an entity is described in terms of the denotations of the entities it is composed of.

Here is a ridiculously simple example.

Consider the following grammar.

digit $\rightarrow$ `'0'` | `'1'` | `'2'` | `'3'` | `'4'` | `'5'` | `'6'` | `'7'` | `'8'` | `'9'`
number $\rightarrow$ *digit*
number $\rightarrow$ *number digit*

Specifying Semantics
Denotational Semantics [2/2]

digit → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

number → *digit*

number → *number digit*

We describe a function *m* that, given a program entity, returns its denotation. In this case, the denotation will be an integer.

- In the first production, for a digit d, $m(d)$ is the usual numeric value. So $m($'0'$) = 0$, $m($'1'$) = 1$, etc.
- In the second production, if a number n is the digit d, then $m(n) = m(d)$.
- In the third production, if a number n expands to a number n_0 followed by a digit d, then $m(n) = 10 \times m(n_0) + m(d)$.

Specifying Semantics
PL Semantics Specification [1/3]

Formal methods for specifying *syntax* have been very successful. Since the late 1970s, virtually all PLs have had a syntax specification in terms of some kinds of grammar.

However, formal *semantics* has been much less successful.

In practice, the semantics of a PL is usually specified in one of two ways.

- Part formally, part informally. Perhaps the semantics of the core constructions of the PL is described informally, while higher level constructions are described using formal semantics, in terms of the core. Example: Haskell.
- Entirely informally, with no formal semantics at all. The meaning/effect of the various constructions is explained. Example: C++.

There are PLs with a complete formal-semantics specification. But this is relatively rare.

Specifying Semantics
PL Semantics Specification [2/3]

In some PLs, it is possible for code to have no specified semantics. Such code is said to have **undefined behavior**.

For example, the C++ Standard does not specify the semantics of code that accesses an array using an out-of-range index.

Consider the following C++ code:

```
k = 99;
int arr[5];  // int array of size 5
arr[k] = 3;  // Index out-of-range - UNDEFINED BEHAVIOR
```

Q. Why is no semantics specified? *Hint. Not because the writers of the Standard made a mistake.*

A. To give compiler writers freedom. Code only needs to work for an in-range index. A compiler writer does not need to worry about what the generated code does for an out-of-range index; no matter what it does, it will be compliant with the Standard.

Specifying Semantics
PL Semantics Specification [3/3]

TO DO

- Look at the official semantics specifications of various programming languages.

Done. We looked at semantics specifications for C++ and Haskell.

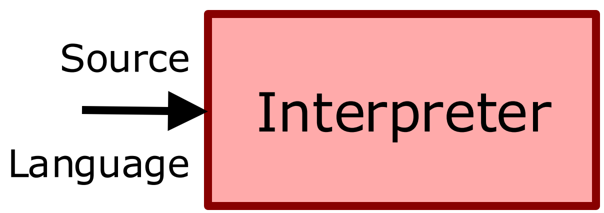
How Interpreters Work

How Interpreters Work
Refresher: Compilation & Interpretation [1/2]

Recall: a **compiler** takes code in one PL (the **source language**) and translates it into code in another PL (the **target language**).

Remember that the target language is *not* necessarily machine language (native code).

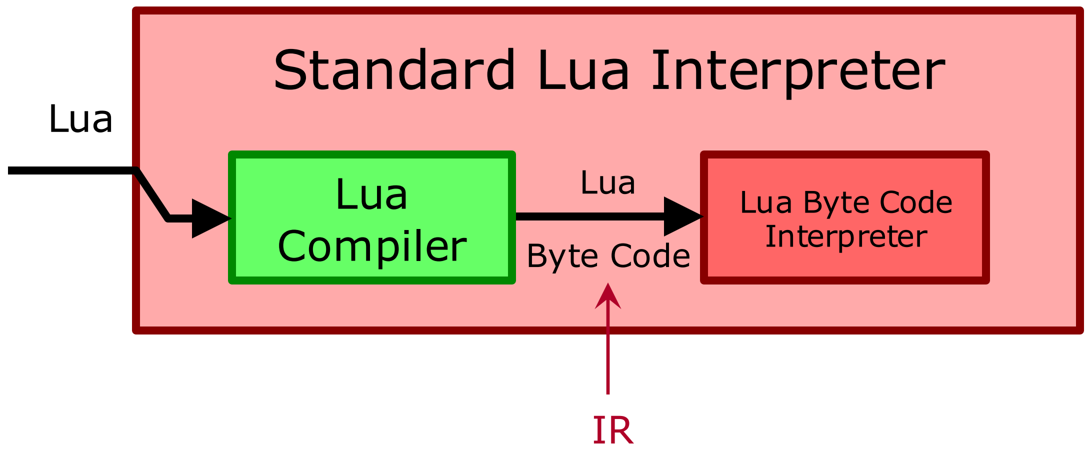
An **interpreter** takes code in some PL and executes it.



Remember:
- **A compiler translates.**
- **An interpreter executes.**

Refresher: Compilation & Interpretation [2/2]

Compilation and interpretation are not mutually exclusive. Many modern interpreters begin by compiling to an **intermediate representation** (**IR**)—perhaps a **byte code**—which is then interpreted directly.



An interpreter or compiler is rarely a monolithic thing. It will be made of separate components (which are composed of components, which are composed of components …).

How Interpreters Work
Kinds of Interpreters

While an interpreter may go through many initial steps (lexical analysis, syntax analysis, byte-code generation, etc.), eventually, the code, in whatever form it ends up in, will need to be executed.

The code module that does the actual execution will usually lie in one of the following four categories:

- **Text-Based Interpreter**
- **Tree-Walk Interpreter**
- **Virtual Machine**
- **JIT**

Next we look at each of these and how it works.

How Interpreters Work
Text-Based Interpreter

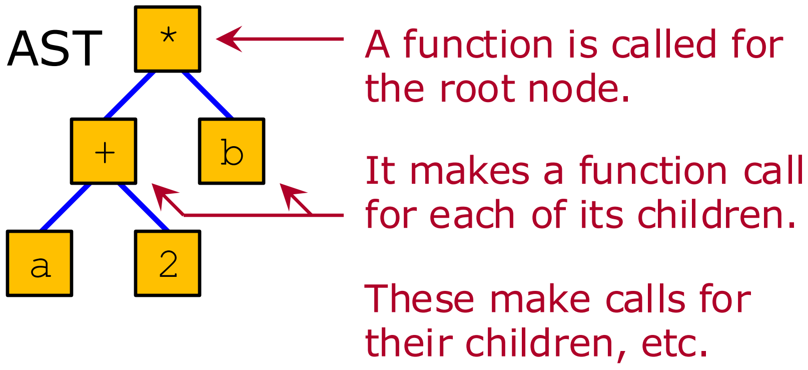
A **text-based interpreter** is one that goes through the high-level source code and executes it directly, line by line, with little or no preprocessing and no intermediate representation used.

Text-based interpreters used to be common. In particular, in the late 1970s, interpreters for the dialects of the BASIC programming language used on early microcomputers were mostly text-based.

However, text-based interpreters generally offer poor performance compared to other methods, and their use has faded. Today they may be used to execute the scripting languages associated with some command-line shells, but not for much else.

How Interpreters Work
Tree-Walk Interpreter

Suppose we parse our source code to obtain an AST. Processing the AST is typically done via mutually recursive functions. A function is called for the root node. It makes a function call for each of its children, and so on.
This is called *walking the tree*.

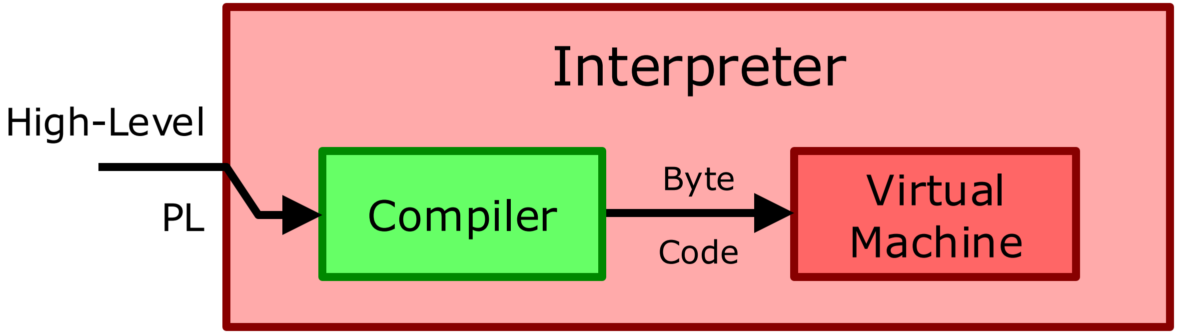


In a **tree-walk interpreter**, these functions do the execution without any further processing.

We know of faster methods; tree-walk interpreters are uncommon. However, they are easy to write. An early release of a PL might include a tree-walk interpreter, with faster interpreters written later. The Ruby PL was handled this way, for example.

How Interpreters Work
Virtual Machine [1/2]

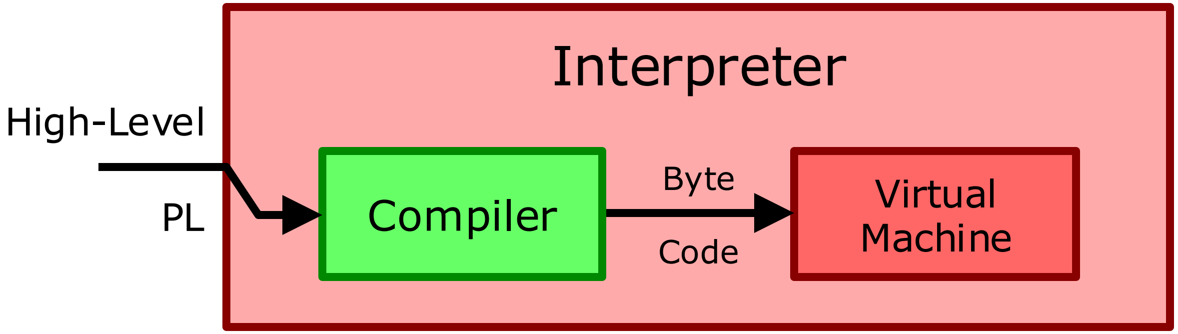
The fastest ways to execute code involve compilation based on the AST. In an interpreter, such a compiler will usually target a machine-language-like programming language designed specifically to be the target PL: a **byte code**.



Code that executes a very low-level PL like a machine language or a byte code is called a **virtual machine** (**VM**). Some VMs emulate processors and portions of computer hardware; these execute some kind of machine language. Other VMs execute some kind of byte code and *might* be used as shown above.

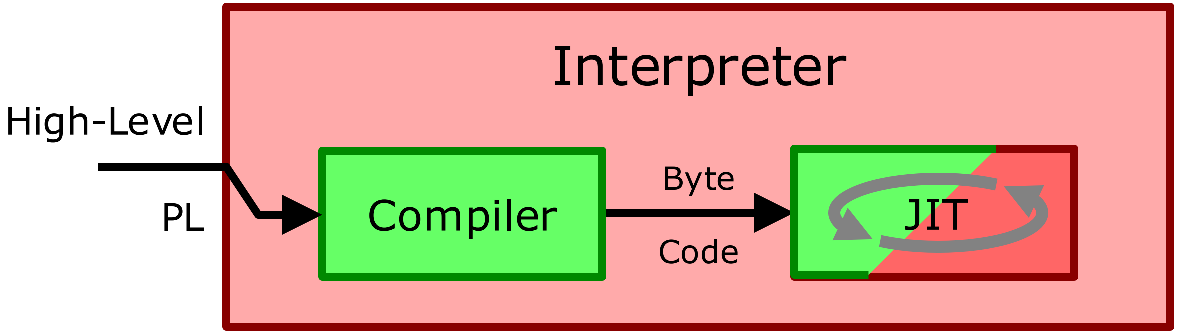
How Interpreters Work
Virtual Machine [2/2]

A virtual machine that is used in a PL interpreter will execute the byte code directly, instruction by instruction.



An interpreter that does compilation to a byte code followed by execution by a VM, as shown above, appears to be the most common kind of interpreter used today. The standard interpreters for Lua, Python, and any number of other programming languages use this design.

Some very fast interpreters execute byte code using a **JIT**—or, more fully, a **JIT** (Just-In-Time) **compiler**. This compiles byte code, usually to machine language, as it executes.

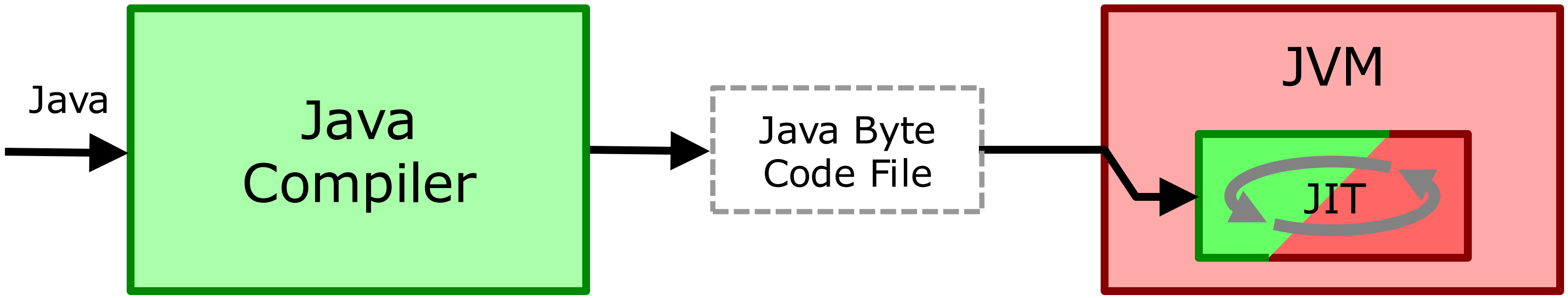


This design is increasingly common. It is used, for example, by the interpreters **LuaJIT** (for Lua) and **PyPy** (for Python).

However, while a well written JIT is fast, it can be labor-intensive to design and code. The decision to write a JIT or something slower depends on whether fast execution is worth the effort.

The JIT was invented in the late 1980s at Xerox's Palo Alto Research Center (PARC), for the programming language **Self**.

In the late 1990s, the Self JIT was the basis of an implementation of the Java Virtual Machine (JVM) called **Hotspot**—which became the default JVM implementation.



All JITs written since have been inspired (at least) by Hotspot.

JITs might seem magical. Let's look at how they work.

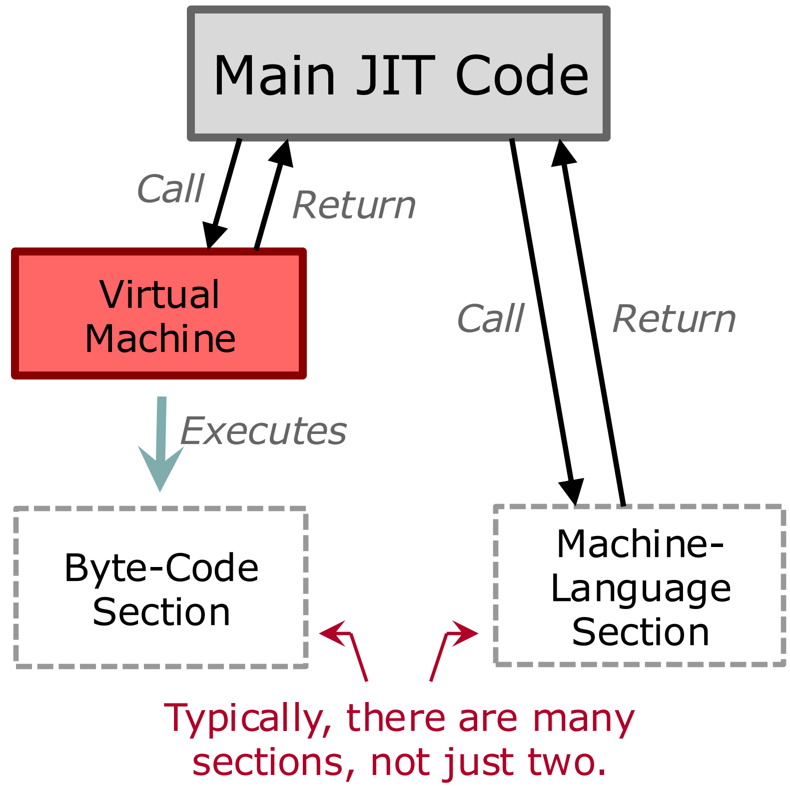
How Interpreters Work
JIT [3/8]

A JIT divides up code to be executed into sections. Ideally, sections involve little flow of control. The JIT tracks whether each section

- is in byte code or has been compiled to machine language, and
- for the latter, how aggressively the section was optimized.

A byte-code section is executed by a VM. A machine-language section can be called like a function.

Between section executions, control returns to the main JIT code. It may compile a byte-code section to machine language, or it may recompile a machine language section with more optimization, or it may do no compilation.

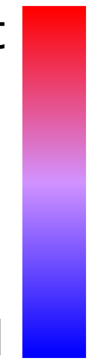


How Interpreters Work
JIT [4/8]

"Just-in-time" actually means compiling at the *optimal* time.

- Compiling early helps realize the benefits sooner.
- But later compilation can use information from code execution—for example, in **profile-based optimizations**, based on what portions of the code spend the most time executing.

Hot



Speed is a high priority

Cold

Speed is a low priority

Each code section is rated **cold** to **hot**, indicating the priority of fast execution. A hot section is:

- more likely to be compiled,
- more likely to be aggressively optimized, and,
- more likely to be *re*-compiled with more optimization.

This is how a *typical* JIT works. Some JITs may be different.

A section's hot/cold rating may be affected by:

- How many times the section has been executed.
- How much time the system has spent executing the section.
- How much of a priority fast execution is at that point.

Here is an example that may help to explain performance improvements with a JIT. Consider the following Lua function.

```
function ff(t)
    t.gg()
end
```

What does function `ff` actually do?

- First, it checks whether `t` is a table. If not, it raises an exception.
- Then it checks whether `t` has a `gg` member—or a metatable with an `__index` member. If not, it raises an exception.
- Having found `gg`, it checks whether it is callable (a function or a table with a metatable having a `__call` member). If not, it raises an exception.
- If all is well, then it calls `gg()`.

That’s a lot! (Issues like this are why dynamic PLs are often slow.)

Part of a program containing function `ff` is shown below.

```
count = 0

x = {}
function x.gg()
    count = count + 1
end
```

```
function ff(t)
    t.gg()
end

for i = 1, 10000000 do
    ff(x)
end
```

What this code ends up doing is repeatedly incrementing an integer `count`. This can be done very quickly. However, function `ff`, in isolation, does not “know” it is doing a fast, easy task.

What can a JIT do about this?

How Interpreters Work
JIT [7/8]

```
count = 0

x = {}
function x.gg()
    count = count + 1
end
```

```
function ff(t)
    t.gg()
end

for i = 1, 10000000 do
    ff(x)
end
```

After, say, 2000 calls to `ff`, the JIT observes that `ff` is very often called with the table `x`, and that the `gg` member of `x` increments `count`, which is an integer. And the relevant section is now hot.

The JIT cannot assume that `ff` will *always* be called with `x`. However, it can assume that `ff` will *probably* be called with `x`—and it can produce code that runs very fast in that case.

JIT [8/8]

The JIT can now compile function `ff` to machine code that acts like the “C” code at right.

```
function ff(t)
    t.gg()
end
```

→

```
void ff(object * t_ptr)
{
    if (t_ptr == &x)
        ++count;
    else …
}
```

The compiled code acts correctly no matter what parameter `ff` is called with. But it is very fast when `ff` is called with `x`. The parameter is usually `x`, so the program executes quickly.

Note, however, that all this only works if the JIT is specifically checking for the things we said it “observes”. What does a JIT need to be checking for? This is a tricky issue.

How Interpreters Work
Summary

There are four main strategies for designing an interpreter. I list them from worst to best performance.

- Do little or no processing of the source code. Execute it line by line, using a **text-based interpreter**. Rare today, except for shells.
- Parse the source code to get an AST. Execute the AST directly, using a **tree-walk interpreter**. Rare today.
- Compile to a byte code. Execute the byte code directly, instruction by instruction, using a **virtual machine**. Very common today.
- Compile to a byte code. Execute the byte code using a **JIT**, which compiles the byte code to machine language as it executes. Somewhat common today, and getting more common.