

Programming Language Syntax Specification

Introduction to Survey of Programming Languages

PL Feature: Compilation & Interpretation

CS 331 Programming Languages
Lecture Slides
Friday, January 24, 2025

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

© 2017–2025 Glenn G. Chappell

Unit Overview

Formal Languages & Grammars

Topics

- ✓ ■ Basic concepts
- ✓ ■ Introduction to formal languages & grammars
- ✓ ■ The Chomsky hierarchy
- ✓ ■ Regular languages
- ✓ ■ Regular languages & regular expressions
- ✓ ■ Context-free languages
 - Programming language syntax specification


Review

A **context-free grammar (CFG)** is a grammar, each of whose productions has a left-hand side consisting of a single nonterminal.

A **context-free language (CFL)** is a language that is generated by some context-free grammar.

CFGs are powerful enough to specify the syntax of most programming languages. They are thus important in **parsing**: determining whether given input (for example, a program) is syntactically correct, and, if so, finding its structure.

We may use a vertical bar to separate the right-hand sides of productions having the same left-hand side.

$S \rightarrow aSa \mid b$  Same as:
 $S \rightarrow aSa$
 $S \rightarrow b$

Review

Context-Free Languages — Parse Trees

Grammar B

$S \rightarrow S+S \mid n$

↖ " + " is a terminal here.

We can represent the structure of a string using a **parse tree**: a rooted tree with one symbol in each node, based on a derivation.

- The root holds the start symbol.
- The symbols a nonterminal is expanded into become its children—left to right, one symbol per tree node.

Here is a parse tree for $n+n+n$, based on the above CFG and derivation.

We can read off the final string by looking at leaves that contain terminal symbols.

Derivation of $n+n+n$

S

S+S

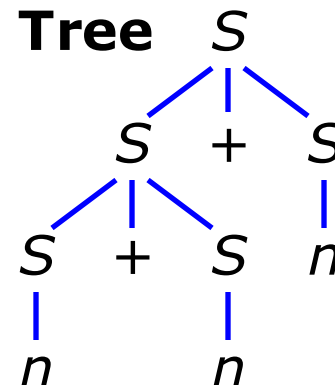
S+S+S

n +S+S

n + n +S

n + n + n

Parse Tree



Review

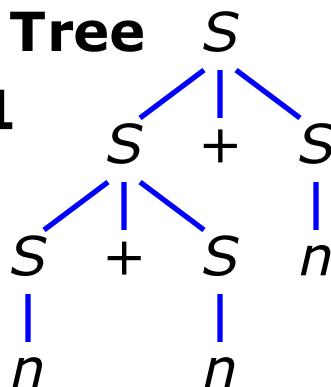
Context-Free Languages — Ambiguity

Grammar B

$S \rightarrow S+S \mid n$

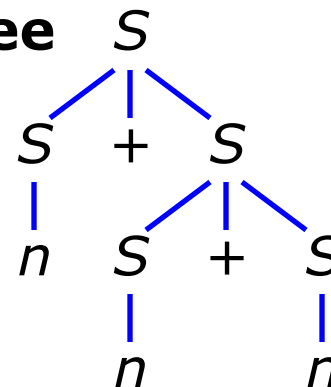
Parse Tree

#1



Parse Tree

#2



A CFG is **ambiguous** if some string has multiple parse trees.

Below is a non-ambiguous CFG that generates the same language and also expresses the left-associativity of "+".

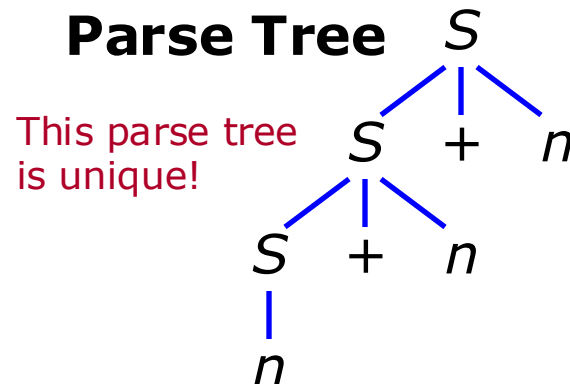
Grammar B'

$S \rightarrow S+n \mid n$

Derivation

S
S+n
S+n+n
n+n+n

Parse Tree

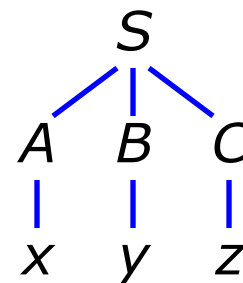


The CFG below generates $\{ xyz \}$. There are multiple derivations.

Grammar D	Leftmost Derivation	Rightmost Derivation	Neither
$S \rightarrow ABC$	<u>S</u>	<u>S</u>	<u>S</u>
$A \rightarrow x$	<u>ABC</u>	<u>ABC</u>	<u>ABC</u>
$B \rightarrow y$	<u>xBC</u>	<u>ABz</u>	<u>AyC</u>
$C \rightarrow z$	<u>xyC</u>	<u>Ayz</u>	<u>Ayz</u>
	xyz	xyz	xyz

When the leftmost nonterminal is expanded each time, we have a **leftmost derivation**. Similarly, **rightmost derivation**.

With the above grammar, there is only one parse tree. Grammar D is *not* ambiguous.



Programming Language Syntax Specification

Programming Language Syntax Specification

Introduction

Now we look at how the syntax of programming languages is specified.

Since the 1970s, the syntax specification of a programming language has generally involved a grammar. This trend was heavily influenced by the release of the *Pascal* programming language by Swiss professor Niklaus Wirth in 1970.

Recall the names we use for various delimiter characters:

Parentheses	()
Brackets	[]
Braces	{	}
Angle brackets	<	>

Programming Language Syntax Specification

Backus-Naur Form [1/4]

Our grammar format (nonterminals are upper-case letters, etc.) is inadequate for specifying the syntax of programming languages.

We want a grammar format that:

- Can deal with terminals involving arbitrary character sets.
- Does not *require* unusual characters (like our “→” and “ ϵ ”).
- Is suitable for use as input to a computer program.
- Allows symbols to have descriptive names (e.g., “for loop”), rather than just single letters.

A solution: **Backus-Naur Form (BNF)**.


- A notation for writing context-free grammars.
- Original idea by John Backus (1959). Revised by Peter Naur (1960).
- BNF has been used to specify the syntax of many programming languages.

Programming Language Syntax Specification

Backus-Naur Form [2/4]

In BNF:

One nonterminal symbol 

- Nonterminals are enclosed in angle brackets: (`<for-loop>`). Inside the angle brackets, the name of the nonterminal must begin with a letter and contain only letters, digits, and hyphens (-).
- The start symbol can vary.
- Terminals are enclosed in quotes: double (`"cat"`) or single (`'"a"'`). *One terminal symbol* 
- Our arrow is replaced by colon-colon-equals: (`::=`).
- The vertical bar (`|`) is used the same way we have used it.
- Epsilon (ϵ) is not used.
- Blanks between nonterminals, terminals, etc., are ignored.
- Each production must lie entirely on a single line.

BNF production for a digit, wrapped (incorrectly!) for lack of space:

```
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5"  
         | "6" | "7" | "8" | "9"
```

Programming Language Syntax Specification

Backus-Naur Form [3/4]

A complete BNF grammar for a U.S. phone number. Productions are (incorrectly!) shown on multiple lines due to lack of space.

```
<phone-number> ::= <area-code> <digit7> | <digit7>
<area-code>    ::= "(" <digit> <digit> <digit> ")"
<digit7>       ::= <digit> <digit> <digit> "-"
                  <digit> <digit> <digit> <digit>
<digit>        ::= "0" | "1" | "2" | "3" | "4" | "5"
                  | "6" | "7" | "8" | "9"
```

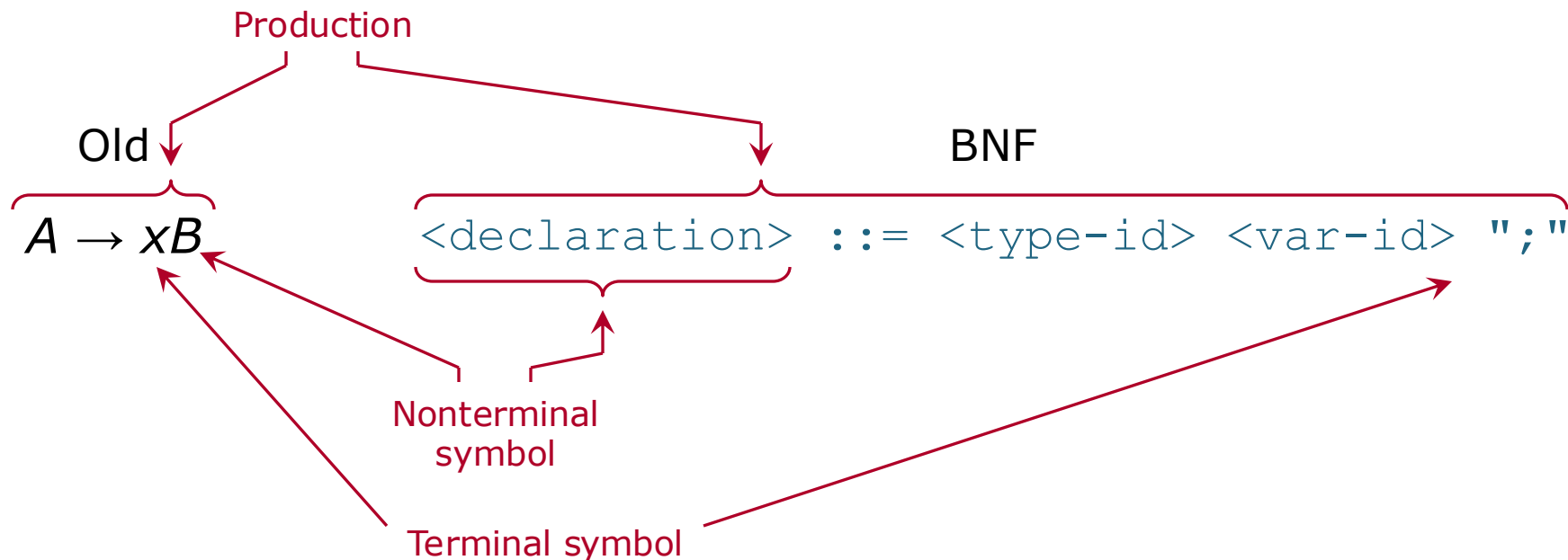
The language generated by the above CFG, with start symbol `<phone-number>`, includes the following strings, among others:

- (907) 474-5736
- 555-1234

Programming Language Syntax Specification

Backus-Naur Form [4/4]

BNF is merely an alternative notation for grammars. All the concepts we have covered are still applicable:



Also: derivation, language generated, whether a grammar is regular, parse tree, ambiguity.

Programming Language Syntax Specification

Extended Backus-Naur Form [1/2]

A number of variations on BNF have been proposed. Some of these are referred to as **Extended BNF (EBNF)**.

EBNF typically includes the following differences from BNF.

- Nonterminals are not enclosed in angle brackets.
- The “:=” is replaced by something shorter, perhaps “=” or “:”.
- A production is allowed to use multiple lines. There is typically an end mark for a production—often a semicolon (;).
- Parentheses may be used for grouping.
- Brackets [...] surround optional sections.
- Braces { ... } surround optional, repeatable sections.

Remember
these!

Note the last two points above. These are used in many kinds of syntax notation (but not BNF!). They correspond, respectively, to “?” and “*” as commonly used in regular expressions.

Programming Language Syntax Specification

Extended Backus-Naur Form [2/2]

Here is the phone-number grammar using a form of EBNF.

```
phone_number = [ area_code ] digit7 ;
area_code    = "(" digit digit digit ")" ;
digit7       = digit digit digit "-"
              digit digit digit digit ;
digit         = "0" | "1" | "2" | "3" | "4" | "5"
              | "6" | "7" | "8" | "9" ;
```

Unlike our earlier BNF grammar, which we had to fudge to get it to fit on the slide, the above is entirely correct.

Programming Language Syntax Specification

Grammars in Practice [1/5]

Today, most programming languages have their syntax specified using a grammar written in something like BNF/EBNF.

Here is one production from a grammar for the C programming language, using the input syntax for the parser generator *Yacc*.

```
compound_statement
: '{' '}'
| '{' statement_list '}'
| '{' declaration_list '}'
| '{' declaration_list statement_list '}'
;
```

Above, terminals are single-quoted. Nonterminals are ordinary words, possibly containing underscores (`_`). The arrow becomes a colon (`:`). A semicolon (`;`) marks the end of a production.

The **lexical structure** of a programming language is about how a program is broken into **lexemes**: identifiers, operators, keywords, etc.

Often (but not always!), lexical structure is specified separately from overall syntax—using a separate grammar, or using some other specification method.

When this is done, a grammar for the overall syntax will have two kinds of terminals:

- Quoted strings indicating exactly what characters must appear.
- *Categories* of lexemes from the lexical-structure specification.

There must be some way of distinguishing the second kind of terminal from a nonterminal. One common method is to place lexeme-category terminals in ALL UPPER CASE.

Programming Language Syntax Specification

Grammars in Practice [3/5]

Here is part of a grammar for an assignment statement in some C-like programming language.

```
assign_stmt = IDENTIFIER assign_op expression ";" ;
assign_op   = "=" | "+=" | "-=" | "*=" | "/=" | "%=" ;
expression  = ...
```

Above, `assign_stmt`, `assign_op`, and `expression` are nonterminals.

`";"`, `"="`, `"+="`, etc., are examples of the first kind of terminal: quoted strings indicating exactly what characters must appear.

`IDENTIFIER` is an example of the second kind of terminal: categories of lexemes from the lexical-structure specification.

Programming Language Syntax Specification

Grammars in Practice [4/5]

BNF, EBNF, and the other grammar conventions we are looking at all have, as one of their goals, being readable by programs.

Grammars that are intended only for humans to read can make use of typographical differences and cut down on punctuation.

For example, here is a production from the 2014 C++ Standard.

selection-statement:

`if (condition) statement`

`if (condition) statement else statement`

`switch (condition) statement`

Nonterminals are in a *slanted* font, while terminals are in a typewriter font. Vertical bars are omitted, with the right-hand sides placed on separate lines. A production has no end mark.

Programming Language Syntax Specification

Grammars in Practice [5/5]

Conclusion. We need standards for representing grammars. These must be used consistently. But exactly what they *are* is less important.

TO DO

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

Done. We looked at syntax specifications for C++, Haskell, Python, and Lua.

Programming Language Syntax Specification

Grammar + Additional Rules — The Dangling “else” Problem [1/2]

It is common that the grammar for a programming language is not *quite* a complete specification of its syntax.

For example, consider the partial C++ grammar two slides back.

```
if (n >= 10) if (n >= 20) ff(); else gg();
```

Which of the following is intended by the above?

```
if (n >= 10)
    if (n >= 20)
        ff();
else
    gg();
```

```
if (n >= 10)
    if (n >= 20)
        ff();
else
    gg();
```

Of course, we do not want to write code like this. Braces are good!

Programming Language Syntax Specification

Grammar + Additional Rules — The Dangling “else” Problem [2/2]

Which of the following is intended?

<pre>if (n >= 10) if (n >= 20) ff(); else gg();</pre>	<pre>if (n >= 10) if (n >= 20) ff(); else gg();</pre>
---	---

We cannot tell. The above correspond to different parse trees; the grammar for C++ is ambiguous! So we add a rule: an “else” is attached to the most recent “if”. The first structure is parsed.

This is the **dangling “else” problem**. It exists in the C family (C, C++, Java, etc.) and in other PLs as well. But it can be avoided. Python, Lua, Haskell, and Scheme do not have the problem.

Programming Language Syntax Specification

Grammar + Additional Rules — Operator Precedence

Another situation in which rules in addition to the grammar are often used involves operator precedence and associativity.

Operator precedence and associativity *can* be specified entirely in the grammar. However, it is often simpler to do so using separate rules.

This is done in the C family (C, C++, Java, etc.). Also see the grammar for Lua.

Unit Overview

The Lua Programming Language

Our second unit: **The Lua Programming Language**.

Topics

- Introduction to survey of programming languages
- PL feature: compilation & interpretation
- PL category: dynamic PLs
- Introduction to Lua
- Lua: fundamentals
- Lua: modules
- Lua: objects
- Lua: advanced flow

After this we will cover **Lexing & Parsing**.

Introduction to Survey of Programming Languages

Introduction to Survey of Programming Languages

From the First Day of Class — Course Overview: Topics

The course material will be divided into eight units:

1. **Formal Languages & Grammars**
2. The Lua Programming Language
 - PL Feature: Compilation & Interpretation
3. **Lexing & Parsing**
4. The Haskell Programming Language
 - PL Feature: Type System
5. The Scheme Programming Language
 - PL Feature: Identifiers & Values
 - PL Feature: Reflection
6. **Semantics & Interpretation**
7. The Prolog Programming Language
 - PL Feature: Execution Model
8. Student Presentations on Programming Languages

Track 1: Syntax &
Semantics of PLs.

Track 2: PL features &
categories, specific PLs.

Introduction to Survey of Programming Languages

Overview

So far, all material we have covered in class has belonged to the **first track**, on syntax and semantics.

Now we switch to the **second track**, beginning a survey of programming languages.

We will look at:

- Features that programming languages can have, and how these features appear in actual programming languages.
- Categories* of programming languages.
- Four specific programming languages, each from a different category: Lua, Haskell, Scheme, and Prolog.

*We will not be doing a complete taxonomy of programming languages. Our categories will be fuzzy, may overlap, and will not include all PLs.

Introduction to Survey of Programming Languages

Some Terminology

A **programming language (PL)** is a notation for specifying computations. A complete specification is called a **program**.

We will occasionally need a term for an arbitrary *thing* in the source for a program: a variable, expression, function, class, etc. The word we will use is **entity**.

An **expression** is an entity that has a value. Below are some C++ expressions.

`-34.5` `x` `(3+g/6)*k` `ff(z)` `"Ostrich"`

Lastly, beware the word *type*. This is a technical term with a specific meaning. Where we might informally say “this type of thing”, let us use the word **kind** or **category**: “this kind of thing”, “this category of thing”.

Introduction to Survey of Programming Languages

Hello-World Programs

Here are hello-world programs in various programming languages.

C++

```
#include <iostream>
using std::cout;

int main()
{
    cout << "Hello, world!\n";
}
```

Lua

```
io.write("Hello, world!\n")
```

Haskell

```
module Main where
main = putStrLn "Hello, world!"
```

Scheme

```
(display "Hello, world!")
(newline)
```

Prolog

```
main :- write('Hello, world!'), nl.
```

PL Feature: Compilation & Interpretation

PL Feature: Compilation & Interpretation Runtime

When a program is **executed**, the computations that it specifies actually occur. The time during which a program is being executed is **runtime**.

These slides are an incomplete summary of the reading "Compilers and Interpreters".

An implementation of a PL will include a **runtime system** (often simply **runtime**): code that assists in, or sometimes performs, execution of a program. This might include low-level I/O, memory management, etc.

Some execution methods never create an executable file or any machine language at all. In such cases, the runtime system will be a separate program that handles all code execution.

PL Feature: Compilation & Interpretation

Compilation [1/2]

A **compiler** takes code in one PL (the **source language**) and transforms it into code in another PL (the **target language**); the compiler is said to **target** this second PL.



Compilers often target **native code** or a **byte code**. However, a compiler can target any PL; for example, there are many compilers that target JavaScript.

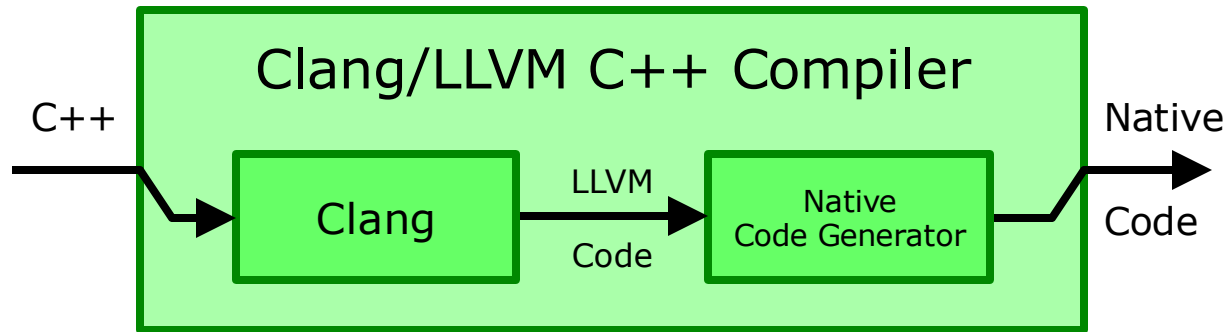
In practice, we usually only use the term “compiler” when:

- the source and target languages differ significantly, and
- The target language is lower-level than the source.

PL Feature: Compilation & Interpretation

Compilation [2/2]

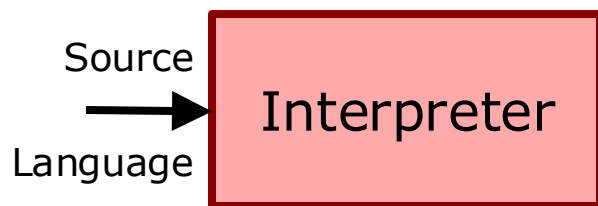
Good compilers proceed in a number of distinct steps. Code is transformed into an **intermediate representation (IR)**, which is then transformed into the ultimate target language.



Arrangements like the above have many advantages, including making it easier to support new source languages and platforms.

PL Feature: Compilation & Interpretation

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

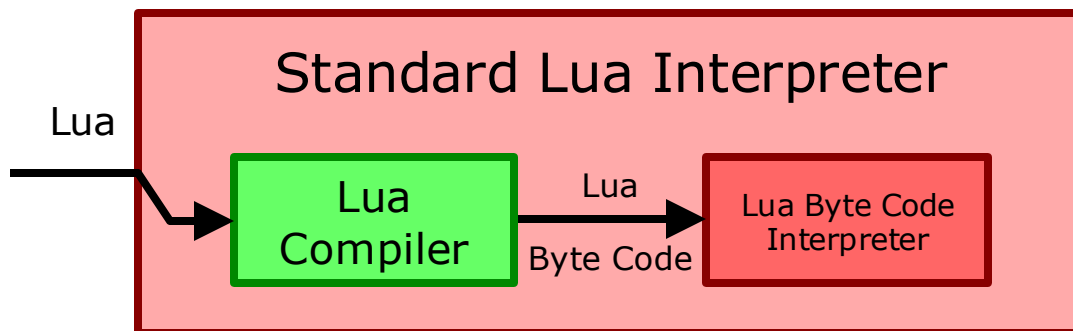


Remember:

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

Two common misconceptions about interpretation:

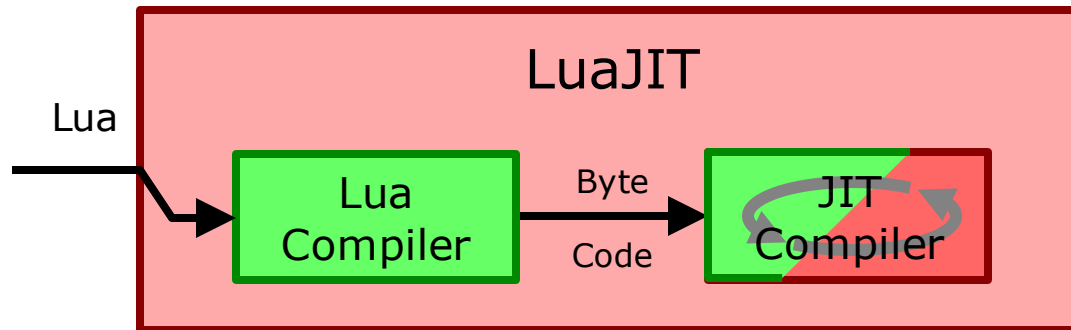
- Interpretation is inherent to a PL.
- Compilation and interpretation are completely separate notions.



PL Feature: Compilation & Interpretation

JIT Compilation

In **Just-In-Time (JIT)** compilation, code is compiled at runtime. A typical strategy is to do static compilation of source code into a byte code. Then execution begins, with the byte code being JIT compiled to native code while the program is executing.



Because the second stage of compilation is done at runtime, information that is only available at runtime can be used—for example, which parts of the code execution spends the most time in.