

Grammars in Practice

Introduction to Survey of Programming Languages

PL Features: Execution I

CS F331 Programming Languages
CSCE A331 Programming Language Concepts
Lecture Slides
Friday, January 25, 2019

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

© 2017–2019 Glenn G. Chappell


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.

Every regular grammar is a CFG. Every regular language is a CFL.

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

Here is a CFG. We can use a vertical bar to separate the right-hand sides of productions with the same left-hand side.

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

This grammar generates the following language.

$\{b, aba, aabaa, aaabaaa, aaaabaaaa, \dots\}$

We can also write this language as follows.

$\{ a^k b a^k \mid k \geq 0 \}$

This is not a regular language.

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.k.a. **concrete syntax tree (CST)**: a rooted tree with one symbol in each node.

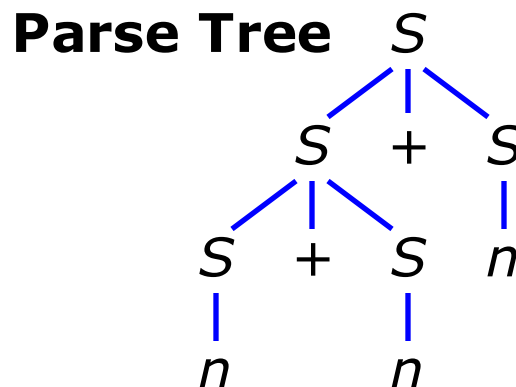
- The root holds the start symbol.
- The symbols a nonterminal is expanded into become its children.

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 the leaves that contain terminal symbols.

Derivation of $n+n+n$

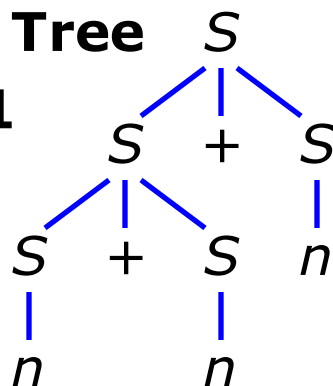
\underline{S}
 $\underline{S}+S$
 $\underline{S}+S+S$
 $n+\underline{S}+S$
 $n+n+\underline{S}$
 $n+n+n$



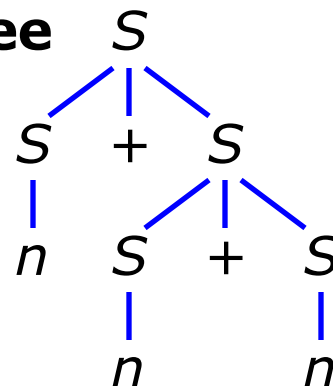
Grammar B

$$S \rightarrow S+S \mid n$$
Parse Tree

#1

**Parse Tree**

#2



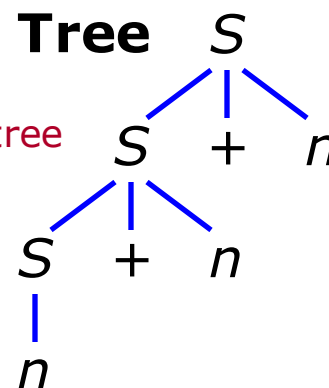
A grammar is **ambiguous** if some string has multiple parse trees. Below is a non-ambiguous grammar that generates the same language and also expresses the left-associativity of “+”.

Grammar B'

$$S \rightarrow S+n \mid n$$
Derivation

$$\begin{aligned} &\underline{S} \\ &\underline{S}+n \\ &\underline{S}+n+n \\ &n+n+n \end{aligned}$$
Parse Tree

This parse tree is unique!



Sometimes ambiguity cannot be eliminated. There are CFLs that are only generated by ambiguous grammars. Such a CFL is **inherently ambiguous**.

Here is a standard example of an inherently ambiguous CFL.

$$\{ a^m b^m c^n d^n \mid m, n \geq 0 \} \cup \{ a^m b^n c^n d^m \mid m, n \geq 0 \}$$

Notes

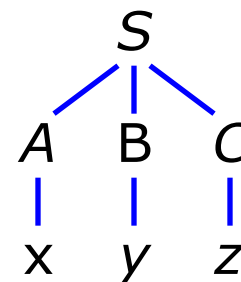
- **Ambiguity** is a property of *grammars* (CFGs).
- **Inherent ambiguity** is a property of *languages* (CFLs).

The CFG below generates only 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.



Grammars in Practice

Backus-Naur Form [1/3]

The CFG format we have been using (upper-case letters are nonterminals, etc.) is inadequate for specifying the syntax of programming languages.

We want a CFG format that:

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

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

- A notation for writing context-free grammars.
- Invented by John Backus and improved by Peter Naur in the late 1950s and early 1960s.
- BNF, or a variation, is used to specify the syntax of many programming languages.

Grammars in Practice

Backus-Naur Form [2/3]

In BNF:

- 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 (`"x"`) or single (`'x'`).
- Our arrow is replaced by colon-colon-equals: (`::=`).
- The vertical bar (`|`) is used the same way we use it.
- Epsilon (ϵ) is not needed.
- Space between nonterminals, terminals, etc., is ignored.

Here is a BNF production specifying what a digit is. Note: a BNF production must lie on a single line; the following is shown on two lines due to lack of space.

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

Grammars in Practice

Backus-Naur Form [3/3]

Here is a complete BNF grammar for a U.S. phone number. Again, productions are 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 includes the following strings, among many others:

- (907)474-5736
- 555-1234

Grammars in Practice

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)**.

Most versions of EBNF include the following differences from BNF.

- Nonterminals are not enclosed in angle brackets.
- The “: :=” is replaced by something shorter, generally “=” or “:”.
- A production ends with a semicolon (;) and may use multiple lines.
- Parentheses may be used for grouping.
- Braces { ... } surround optional, repeatable sections.
- Brackets [...] surround optional sections.

Important!

The last two points above are particularly noteworthy. They apply to virtually all modern grammar notations, and we will use them when specifying PL syntax. Note that they correspond, respectively, to “*” and “?” as used in regular expressions.

Grammars in Practice

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 a bit to get it to fit on the slide, the above is entirely correct.

Grammars in Practice

Programming-Language Syntax [1/4]

Variations on BNF/EBNF are used to write formal grammars for most programming languages.

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 '}'
;
```

Terminals are 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.

It is common for lexical structure to be 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-class terminals in ALL UPPER CASE.

Grammars in Practice

Programming-Language Syntax [3/4]

Here is part of an (overly simple) grammar for a C++ assignment statement.

```
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.

Human-readable grammars can make use of typographical differences and cut down on punctuation. For example, here is a production from the grammar for C++ in the 2011 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 various possible right-hand sides placed on separate lines.

Conclusion. We need notational conventions. If a CFG is input to a program, then conventions must be rigidly enforced. However, it is not so important exactly what the conventions are.

Introduction to Survey of Programming Languages [1/3]

So far, all class material 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.
- Five specific programming languages: Lua, Haskell, Forth, Scheme, and Prolog.

Introduction to Survey of Programming Languages [2/3]

Here are *Hello World* programs in various programming languages.

C++

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

int main()
{
    cout << "Hello, world!"
        << endl;
    return 0;
}
```

Lua

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

Haskell

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

Forth

```
." Hello, world!" cr
```

Scheme

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

Prolog

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

Introduction to Survey of Programming Languages [3/3]

Before we go on, some terminology.

As we have mentioned before, an **expression** is something that has a value. Below are some C++ expressions.

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

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

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**: “this kind of thing”.

PL Features: Execution I

Runtime [1/2]

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

An implementation of a PL will include a **runtime system** (often simply **runtime**): code that assists in, or sometimes performs, execution of a program.

One service that is typically provided by a runtime system is **memory management**.

- In C++, `new` and `delete` call code in the runtime system.
- PLs like Java, Python, and Lua have runtime systems that do **garbage collection**: cleaning up no-longer-used memory blocks, so that a program need not do explicit deallocation.

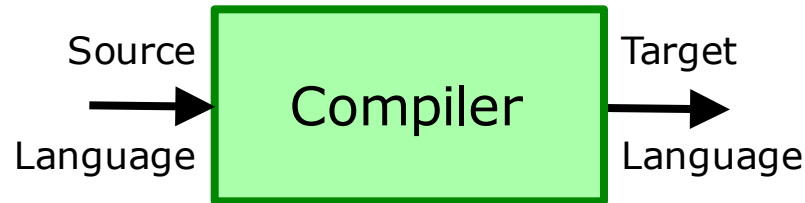
Executable files will generally include the runtime system—or at least the parts of it that a program uses.

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

PL Features: Execution I

Compilation [1/4]

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.



A compiler might target **native code**—the machine language directly executed by a computer's processor—or it might target some other PL. The actual code that a compiler outputs is **object code**.

Some PLs are intended solely as target languages and are not aimed at readability for humans; such a PL is a **byte code**.

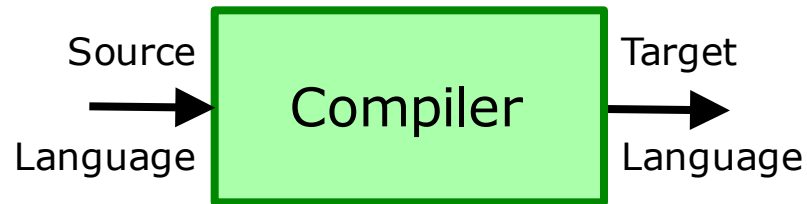
- For example, Java programs are usually compiled to **Java byte code**, which can be executed by a runtime system called the **Java Virtual Machine (JVM)**.

PL Features: Execution I

Compilation [2/4]

In practice, we usually only use the term “compiler” when all of the following are true.

- The source and target languages differ significantly.
- The target language is lower-level than the source.
- The transformation is done with execution as the goal.



Software that transforms code in some PL to code in a very similar PL might be called a **preprocessor** or an **assembler**.

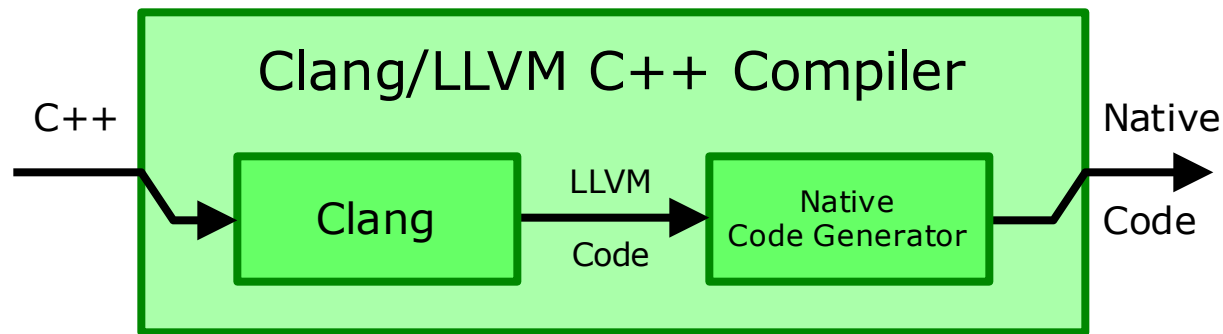
Software that compiles code in some PL to code in another PL that operates at roughly the same level of abstraction might be called a **transcompiler** or **transpiler**.

PL Features: Execution I

Compilation [3/4]

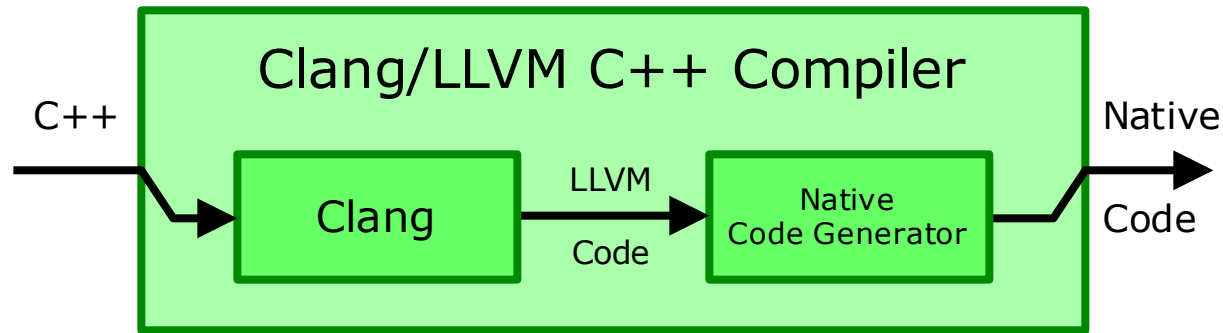
Good compilers proceed in a number of distinct steps. Code is transformed into an **intermediate representation (IR)**, which is then be transformed into the ultimate target language.

For example, the C++ compiler shipped with Apple's Xcode IDE is called **Clang**. Clang itself does not target native code, but rather the machine-independent IR specified by the **LLVM** (Low-Level Virtual Machine) project.



PL Features: Execution I

Compilation [4/4]



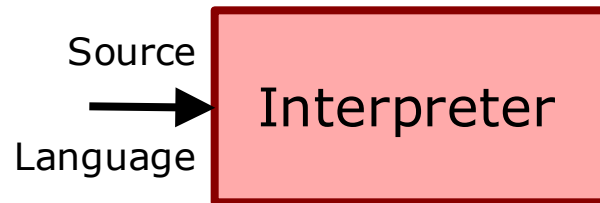
Arrangements like that above have important advantages.

- To add a new source PL to the above system, we only need to write a compiler for this PL that targets LLVM code. Native code generators already exist for all common platforms.
- Similarly, to target a new platform, we only need to write one code generator. Then all supported source PLs can be compiled with our new platform as the target.

PL Features: Execution I

Interpretation [1/3]

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



There are two common misconceptions about interpretation.

The first is that interpretation is inherent to a PL—that there are interpreted PLs and non-interpreted PLs, forever distinct.

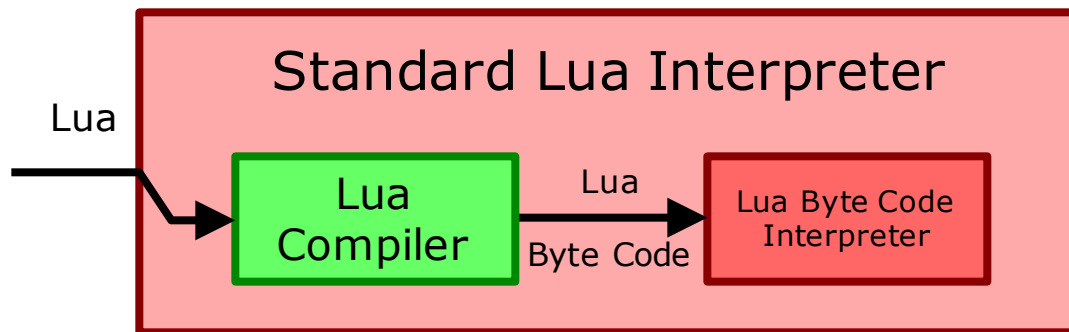
Certainly, there are PLs that are usually interpreted, but one can still write a compiler for one of these, targeting native code. Similarly, the usual C++ build process involves compilation that produces an executable file. But one can write a C++ interpreter; indeed, these exist.

PL Features: Execution I

Interpretation [2/3]

The second misconception is that compilation and interpretation are completely separate notions. But many interpreters actually begin with a compilation step.

For example, Lua is usually interpreted. The standard Lua interpreter begins by compiling Lua code to a low-level PL called **Lua Byte Code**. This byte code is then interpreted directly.



A process like that above is common. To describe the operation of the Python interpreter in the standard Python distribution (CPython), replace every instance of "Lua" above with "Python".

Some interpreters allow code to be executed in an **interactive environment**. A user can type in a single statement, expression, or other chunk of code—whatever is appropriate for the PL. This is executed, and its output, if any, is printed. Then the user is prompted for more code.

Such an environment is sometimes called a **REPL** (Read-Eval-Print Loop), a term that originated with the Lisp family of PLs.

Some code transformations require information that is only available at runtime. Examples of these are **profile-based optimizations**, which transform code based on what portions of the code spend the most time executing. A newer kind of compiler is actually able to perform such transformations.

It may seem impossible to transform code while it is executing, but such **dynamic compilation** is actually becoming common. The resulting concurrent compilation & execution is termed **Just-In-Time (JIT)** compilation, and a program that does it is a **JIT compiler**, or sometimes simply a **JIT**.

PL Features: Execution I

JIT Compilation [2/2]

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.

This is, in fact, the strategy used by **LuaJIT**, a Lua interpreter that does JIT compilation.

