

# Programming Languages

CSCI-GA.2110-001

Spring 2023

Dr. Cory Plock

# What this course is

- A study of programming language paradigms
  - ◆ Imperative
  - ◆ Functional
  - ◆ Logical
  - ◆ Object-oriented
- Tour of programming language history & roots.
- Introduction to core language design & implementation concepts.
- Exposure to languages/paradigms you may not have used before.
- Exploration of implementation issues, including performance considerations.
- Understanding and appreciation of language standards.
- Enable you to more quickly learn and understand new languages.

# What this course isn't

- A learn-to-program course.
- Learn the hottest new languages
  - ◆ We study languages not because of their popularity or marketability but due to their contributions to the field.
- Become an expert in 10 new languages.
- Project-based software development course.
- A software engineering course.
- A compiler course.

# Introduction

The main themes of programming language design and use:

- Paradigm (model of computation)
- Expressiveness
  - ◆ control structures
  - ◆ abstraction mechanisms
  - ◆ types and their operations
  - ◆ tools for programming in the large
- Ease of use: writeability / readability / maintainability
- Portability

# Language as a tool for thought

- Role of language as a communication vehicle among programmers can be just as important as ease of writing
- All general-purpose languages are *Turing complete* (They can compute the same things)
- But languages can make expression of certain algorithms difficult or easy.
  - ◆ Try multiplying two Roman numerals
- Idioms in language A may be useful inspiration when writing in language B.

# Idioms

- Copying a string `q` to `p` in C:

```
while (*p++ = *q++) ;
```

- Removing duplicates from the list `@xs` in Perl:

```
my %seen = ();  
@xs = grep { ! seen{ _ }++; } @xs;
```

- Computing the sum of numbers in list `xs` in Haskell:

```
foldr (+) 0 xs
```

Is this natural?     *It is if you're used to it*

# Some programming paradigms

- *Imperative (von Neumann)*: **Fortran, Pascal, C, Ada**
  - ◆ programs have mutable storage (state) modified by assignments
  - ◆ the most common and familiar paradigm
- *Functional (applicative)*: **Scheme, Lisp, ML, Haskell**
  - ◆ functions are first-class values
  - ◆ *side effects* (e.g., assignments) discouraged
- *Logical (declarative)*: **Prolog, Mercury**
  - ◆ programs are sets of assertions and rules
- *Object-Oriented*: **Simula 67, Smalltalk, C++, Ada95, Java, C#**
  - ◆ data structures and their operations are bundled together
  - ◆ inheritance
- *Quantum*: **QCL, Q, Q#, qGCL**
  - ◆ performs operations on data using quantum bits (“qubits”)
  - ◆ utilizes quantum properties such as superposition and entanglement

# Choice of Languages

- Survey on the actual motivators for language choice:
  - ◆ “That’s what everybody happens to know.”
  - ◆ “It’s the hottest new language with cool features.”
- What it *should* be:
  - ◆ “It’s the best language for the job.”
  - ◆ “It contains abstractions we find useful for solving the problem.”
  - ◆ “It is the most readable language for programs in this domain.”
- Fun fact: there are over 3,000 languages in the world. Not a single one of them has been proven to decrease defect rates.



# Spectrum of Languages

- High level languages are an abstraction: an interface between humans and the computer.
- There are differing levels of abstraction, known as the *program level*. (Halstead)
- Low level languages: you tell the language how to solve the problem. (**Assembly, C, Fortran**)
- High level languages: you tell the language what to do, not how to do it. (**ML, Setl, Prolog**)
- Hybrid languages: a little of both (**Ada, C++, Java, C#**)

# Beginnings

- Before high level languages, programs were written in *assembly*.
  - ◆ Hardware-specific.
  - ◆ Not easily ported.
  - ◆ Repetition of the same patterns.
  - ◆ More difficult to reuse code.
  - ◆ Great effort for even simple algorithms.
  - ◆ High probability of programming error.
  - ◆ Chance of wearing out or even damaging hardware.

# Beginnings

## ■ FORTRAN

- ◆ Invented by John Backus et al., released in 1957.
- ◆ First successful high-level programming language.
- ◆ Primary use: scientific computing and mathematics.
- ◆ Example:

`A = C + D`

## ■ COBOL

- ◆ Designed by committee, released late 1960.
- ◆ Common or Business-Oriented Language.
- ◆ Data processing, business, finance, administrative systems.
- ◆ Introduced structures.
- ◆ Example:

`ADD C TO D GIVING A`

# Beginnings (continued)

## ■ ALGOL

- ◆ Invented by a group of European & American computer scientists, released in 1958.
- ◆ Popularized many PL concepts still in use today.
  - BNF
  - Compound statements using blocks
  - case statement
  - Call-by-reference
  - Concurrency
  - Orthogonality
- ◆ Was not a commercial success (e.g., no standard I/O).

```
IF Ivar > Jvar THEN Ivar ELSE Jvar FI := 3;
```

# Language standards

Developed by working groups of standards bodies (ANSI, ISO).

- Main goal: define the language.
- Pro: Discourages language “flavors” (like LISP), increases portability.
- Con: Places creative freedom in the hands of a few people.
- Major compiler manufacturers generally align to the standards.
- Defines syntactic and semantic correctness (sometimes partially).
- Enforcement is often left to individual compiler implementations.
- Undefined behavior may not be detected. Decreases portability.

Example 1 (C++):

```
int x;  
int y = x + 2; // x is undefined
```

Example 2 (C++):

```
cout << (INT_MAX+1);
```

See also: [this research paper](#)

# Language libraries

The programming environment may be larger than the language.

- The predefined libraries are *indispensable* to the proper use of the language, *and its popularity*.
- The libraries are defined in the language itself, but they have to be internalized by a good programmer.

Examples:

- C++ standard template library
- Java Swing classes
- Ada I/O packages

# Syntax and semantics

- Syntax refers to external representation:
  - ◆ Given some text, is it a well-formed program?
- Semantics denotes meaning:
  - ◆ Given a well-formed program, what does it mean?
  - ◆ Often depends on context (e.g. C++ keyword **const**, operator **&**).

The division is somewhat arbitrary.

- Similar looking constructs in different languages often have different meanings (e.g. **virtual**)

# Compilation overview

Major phases of a compiler:

1. lexer: text  $\longrightarrow$  tokens
2. parser: tokens  $\longrightarrow$  parse tree
3. semantic analyzer: parse tree  $\longrightarrow$  abstract syntax tree
4. intermediate code generation
5. optimization (machine independent): local & global redundancy elimination, loop optimization
6. target code generation
7. optimization (machine dependent): instruction scheduling, register allocation, peephole optimization

Some languages (Java) perform some steps at compile time and others at runtime.



# Grammars

A *grammar*  $G$  is a tuple  $(\Sigma, N, S, \delta)$

- $\Sigma$  is the set of *terminal* symbols (alphabet)
- $N$  is the set of *non-terminal* symbols
- $S$  is the distinguished non-terminal: the root symbol
- $\delta$  is the set of rewrite rules (productions) of the form:

$$ABC \dots ::= XYZ \dots$$

where  $A, B, C, X, Y, Z$  are terminals and non terminals.

- The symbol  $\epsilon$  represents the *empty string*. Epsilon transition:  $A ::= \epsilon$  means “nonterminal  $A$  rewrites to no terminals or non-terminals.”
- The *language* is the set of sentences containing **only** terminal symbols that can be generated by applying the rewriting rules starting from the root symbol (let's call such sentences *strings*)

# BNF for context-free grammars

(BNF = Backus-Naur Form)

Does not add expressiveness to the language—for convenience only. Some conventional abbreviations:

- alternation:  $\text{Symb} ::= \text{Letter} \mid \text{Digit}$
- sequencing:  $\text{Id} ::= \text{Letter Symb}$

EBNF = Extended Backus-Naur Form. Encompasses everything above, plus:

- repetition:  $\text{Id} ::= \text{Letter} \{\text{Symb}\}$   
or we can use a Kleene star:  $\text{Id} ::= \text{Letter Symb}^*$   
for one or more repetitions:  $\text{Int} ::= \text{Digit}^+$
- option:  $\text{Num} ::= \text{Digit}^+ [\text{Digit}^*]$
- grouping:  $\text{SignedNum} ::= ('+'|-') \text{Num}$

Note: need convention for metasymbols – what if “|,” “[,” “],” “+,” etc. are in the language?

# Partial Java Grammar

```
<typedekl>      ::= type <typedeflist>
<typedeflist>   ::= <typedef> [ <typedeflist> ]
<typedef>       ::= <typeid> = <typespec> ;
<typespec>      ::= <typeid> |
                    <arraydef> | <ptrdef> | <rangedef> |
                    <enumdef> | <recdef>

<typeid>        ::= <ident>
<arraydef>      ::= [ packed ] array '[' <rangedef> ']' of <typeid>
<ptrdef>        ::= ^ <typeid>
<rangedef>      ::= <number> .. <number>
<number>        ::= <digit> [ <number> ]
<enumdef>       ::= ( <idlist> )
<idlist>        ::= <ident> { , <ident> }
<recdef>        ::= record <vardecllist> end ;
```

# The Chomsky hierarchy

## ■ Regular grammars (Type 3)

- ◆ all productions can be written in the form:  $N ::= TN$
- ◆ one non-terminal on left side; at most one on right
- ◆ generally used for scanners

## ■ Context-free grammars (Type 2)

- ◆ all productions can be written in the form:  $N ::= XYZ$
- ◆ one non-terminal on the left-hand side; mixture on right
- ◆ most major programming languages

## ■ Context-sensitive grammars (Type 1)

- ◆ number of symbols on the left is no greater than on the right
- ◆ no production shrinks the size of the sentential form
- ◆ used for parts of C++, but otherwise rarely used

## ■ Type-0 grammars

- ◆ no restrictions

# Regular grammar example

A grammar for floating point numbers:

$$\text{Float} ::= \text{Digits} \mid \text{Digits} . \text{Digits}$$
$$\text{Digits} ::= \text{Digit} \mid \text{Digit Digits}$$
$$\text{Digit} ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

A regular expression for floating point numbers:

$$(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^+ (.(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^+)?$$

Perl offer some shorthands:

$$[0-9]^+(\.[0-9]^+)?$$

or

$$\backslash d^+(\backslash . \backslash d^+)?$$

# Regular expressions

Regular grammars can be used to *generate* regular languages. Regular expressions can be used to *accept* regular languages.

We say that a regular expression  $R$  denotes the language  $\llbracket R \rrbracket$ .

Basic regular expressions:

- $\epsilon$  denotes  $\emptyset$
- a character  $x$ , where  $x \in \Sigma$ , denotes  $\{x\}$
- (sequencing) a sequence of two regular expressions  $RS$  denotes  $\{\alpha\beta \mid \alpha \in \llbracket R \rrbracket, \beta \in \llbracket S \rrbracket\}$
- (alternation)  $R|S$  denotes  $\llbracket R \rrbracket \cup \llbracket S \rrbracket$
- (Kleene star)  $R^*$  denotes the set of strings which are concatenations of zero or more strings from  $\llbracket R \rrbracket$
- parentheses are used for grouping

Shorthands:

- $R^? \equiv \epsilon|R$
- $R^+ \equiv RR^*$

# Regular expressions

The following are not strictly regular expressions, but rather notational conveniences:

- $\cdot \equiv \text{any } \alpha \in \Sigma$  (“any character”)
- Character classes:
  - ◆  $[abc] \equiv (a|b|c)$
  - ◆  $[\^abc] \equiv \Sigma \setminus \{a, b, c\}$
  - ◆  $[0-9] \equiv (0|1|2|3|4|5|6|7|8|9)$ .
  - ◆ Similarly with a-z and A-Z.

Expressions can be *greedy* or *lazy*:

- Greedy: match as much as possible (Flex)
- Lazy: match as little as possible

Example:  $f.+1$  on input string “fellow”

- Greedy: will match “fell”
- Lazy: will match “fel”

# Lexical Issues

Lexical: formation of words or tokens.

- Described (mainly) by regular grammars
- Terminals are characters. Some choices:
  - ◆ character set: ASCII, Latin-1, ISO646, Unicode, etc.
  - ◆ is case significant?
- Is indentation significant?
  - ◆ Python, Occam, Haskell

Example: identifiers

$$\text{Id} ::= \text{Letter IdRest}$$
$$\text{IdRest} ::= \epsilon \mid \text{Letter IdRest} \mid \text{Digit IdRest}$$

Missing from above grammar: limit of identifier length



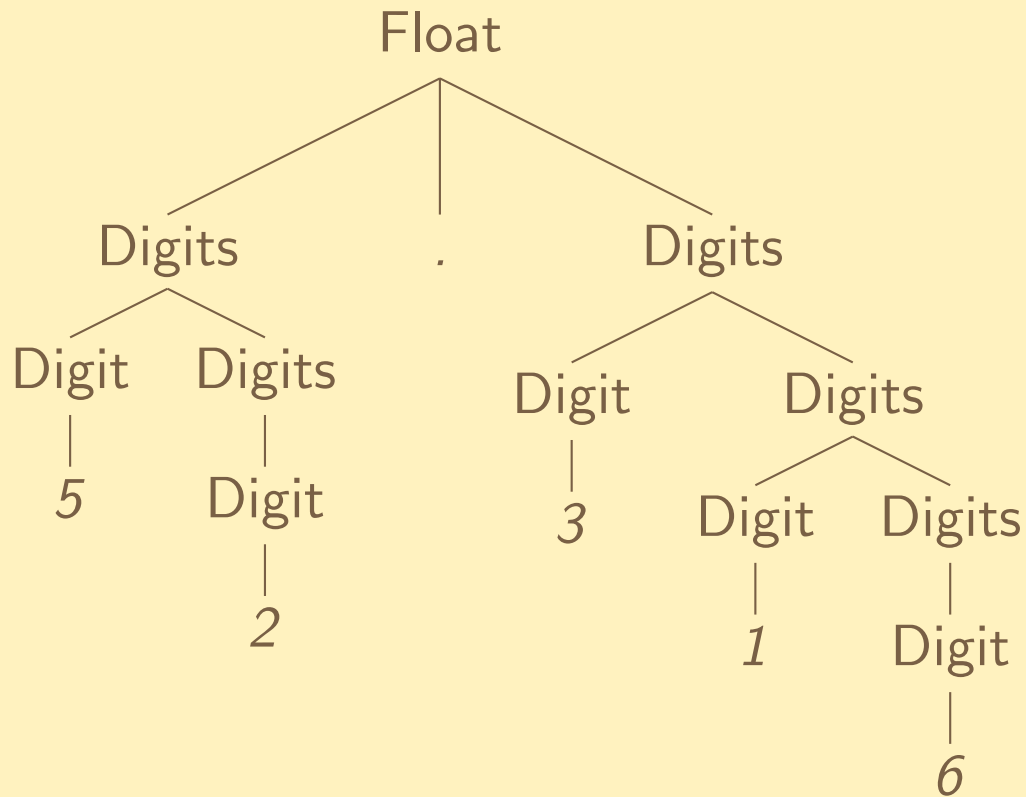
# Parse trees

A parse tree describes the grammatical structure of a sentence

- root of tree is root symbol of grammar
- leaf nodes are terminal symbols
- internal nodes are non-terminal symbols
- an internal node and its descendants correspond to some production for that non terminal
- top-down tree traversal represents the process of generating the given sentence from the grammar
- construction of tree from sentence is *parsing*

# Parse Tree Illustration

The parse tree for input string 52.316 on the grammar below from slide 20:



Float ::= Digits | Digits . Digits

Digits ::= Digit | Digit Digits

Digit ::= 0|1|2|3|4|5|6|7|8|9

# Ambiguity

If the parse tree for a sentence is not unique, the grammar is *ambiguous*:

$$E ::= E + E \mid E * E \mid \text{Id}$$

Two possible parse trees for “A + B \* C”:

- $((A + B) * C)$
- $(A + (B * C))$

One solution: rearrange grammar:

$$\begin{aligned} E &::= E + T \mid T \\ T &::= T * \text{Id} \mid \text{Id} \end{aligned}$$

Harder problems – disambiguate these (courtesy of Ada):

- $\text{function\_call} ::= \text{name} (\text{expression\_list})$
- $\text{indexed\_component} ::= \text{name} (\text{index\_list})$
- $\text{type\_conversion} ::= \text{name} (\text{expression})$

# Precedence

Consider the expression  $5 + 2 * 3$ .

- In mathematics, we say operator  $*$  has *precedence* over operator  $+$ .
- This means evaluate the expression as:  $5 + (2 * 3)$ , not  $(5 + 2) * 3$ .
- Do not confuse precedence with order of evaluation.
  - ◆ Precedence indicates grouping.
  - ◆ Groups can be evaluated in any order.

Precedence can be specified in a couple ways:

- Write the precedence rules directly into the grammar. Rules for higher precedence operators will tend to be deeper in the parse tree than other rules. (see “calc1” example)
- Write an ambiguous grammar (i.e. make  $+$  and  $*$  the same level of precedence), specify operator precedence separately. Parser generators like Bison offer this as a convenience. (see “calc2” example)

# Associativity

Consider the expression  $5 + 2 + 3$ .

We know that  $5 + (2 + 3)$  yields the same mathematical result as  $(5 + 2) + 3$ , but  $5 - (2 - 3)$  is different than  $(5 - 2) - 3$ .

*Associativity* tells the parser what to do with operators at the *same* level of precedence.

Some options:

- Use *left associativity*:  $((5 + 2) + 3)$
- Use *right associativity*:  $(5 + (2 + 3))$

# Dangling else problem

Consider:

$$S ::= \text{if } E \text{ then } S$$
$$S ::= \text{if } E \text{ then } S \text{ else } S$$

The sentence

$$\text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2$$

is ambiguous (Which then does else  $S_2$  match?)

Solutions:

- Pascal rule: else matches most recent if
- grammatical solution: different productions for balanced and unbalanced if-statements
- grammatical solution: introduce explicit end-marker

The general ambiguity problem is unsolvable

# Scanners and parsers

- **Scanners** (or *tokenizers*) read input, identify, and extract small input fragments called tokens.
  - ◆ Identifiers
  - ◆ Constants
  - ◆ Keywords
  - ◆ Symbols: (, ), [, ], !, =, !=, etc.
- **Parsers** accept a grammar and tokens as input and attempt to construct a parse tree.
  - ◆ **LL** (or: recursive descent, predictive) parsers are depth-first, begin at the start symbol, predict the next rewrite rule, and recurse on it. Implementation: “by hand” or table-driven.
  - ◆ **LR** (or: bottom-up) parsers find LHS non-terminals that match the input tokens already seen. Normally faster in production compilers (exception: gcc). Implementation: almost always table-driven.

Relationships: **LL**  $\subset$  **LR**  $\subset$  **CF**

# LL Parsers

- LL stands for: left-to-right, leftmost derivation.
- Also known as top-down, recursive descent, or predictive parsers.
- Begin at the root symbol.
- For each RHS non-terminal, decide which rewrite rule to use (if more than one).
- Ideal: only one rule to choose from. Deterministic.
- More common: multiple rules exist. Non-deterministic.
- Error: no rule exists.
- Resolving non-determinism: decide based on the next token.
- If cannot resolve based on next, “look ahead” up to  $k$  tokens.
- We refer to parsers with  $k$  look ahead as LL( $k$ ) parsers.
- Note 1: it may be possible to modify the grammar to an equivalent grammar with a smaller (or no) lookahead requirement.
- Note 2: most production LL parsers are LL(1).



# Problems with LL parsing

**Left recursion:** a grammar is left-recursive if there exists non-terminal  $A$  such that  $A \Rightarrow^+ A \alpha$  for some  $\alpha$ . Example:

$$\begin{aligned} \text{id\_list} &\Longrightarrow \text{id\_list\_prefix} ; \\ \text{id\_list\_prefix} &\Longrightarrow \text{id\_list\_prefix} , \text{id} \\ &\Longrightarrow \text{id} \end{aligned}$$

**Common prefixes:** if there exists a non-terminal  $A$  and terminal  $b$  such that there exist rules  $R_1 : A \Rightarrow^* b \dots$  and  $R_2 : A \Rightarrow^* b \dots$

$$\begin{aligned} \text{stmt} &\Longrightarrow \text{id} := \text{expr} \\ &\Longrightarrow \text{id} ( \text{argument\_list} ) \end{aligned}$$

Solution to both issues: rewrite the grammar.

(Examples courtesy of Scott. See pp. 79-80 for solutions).

# LR Parsers

- Stands for: left-to-right, rightmost derivation.
- Also known as bottom up or shift-reduce parsers.
- With LR parsers, the bottom of the parse tree is built first.
- If the root symbol is reachable, the parser accepts the input.
- Main data structure is a stack.
- Main operations are: **shift** and **reduce**.
- LR parsers *shift* tokens to the stack.
- Each time a token is shifted, the stack is checked to see if the tokens match the right side of a rule.
- If so, we replace the tokens on the stack with the left-hand side of the rule. This is the *reduce* step.
- If the stack is empty when the input is fully read, the input is accepted.
- If the stack is non-empty, there exist tokens with no corresponding rule: error.

# Problems with LR parsing

**Shift-reduce conflicts:** when the choice of shifting or reducing is non-deterministic. Example:

$$\begin{aligned}\text{if\_stmt} &\implies \text{IF expr THEN stmt} \\ &\implies \text{IF expr THEN stmt ELSE stmt}\end{aligned}$$

Suppose the parser has read IF expr THEN stmt (but not yet ELSE)  
The parser could shift ELSE **or** reduce the above to if\_stmt—there isn't enough information for it to properly choose.

Solutions: rewrite grammar (see previous slides), introduce lookahead.

**Reduce-reduce conflicts:** when there are multiple possible reductions (e.g. the RHS is the same). Example:

$$\begin{aligned}\text{if\_stmt} &\implies \text{IF expr THEN stmt} \\ \text{if\_stmt\_2} &\implies \text{IF expr THEN stmt}\end{aligned}$$

# Creating scanners and parsers

- **Lex** (or **Flex**) is a lexical analyzer generator.
  - ◆ Input: rules containing regular expressions.
  - ◆ Output: C code. Can be compiled into a standalone lexical analyzer or integrated into a parser.
- **Yacc** (or **Bison**) is a parser generator.
  - ◆ Input: Context-free grammar and Lex generated source code (optional).
  - ◆ Output: An LR parser.

Another popular parser generator is ANTLR (Another Tool for Language Recognition).