

# Programming Paradigms

## Syntax (Part 1)

**Prof. Dr. Michael Pradel**

**Software Lab, University of Stuttgart**

**Summer 2022**

# Motivation

---

- **Goal:**

  - **Specify a programming language**

    - What code is part of the language?
    - What is the meaning of a piece of code?

- **Important for both developers and tools**

- **In contrast: Natural languages not formally specified**

# Syntax vs. Semantics

---

**Structure** of  
code

**Meaning** of  
code

**Example:**

**Grammar to define a language: Could mean**

**digit**  $\rightarrow$  0 | 1 | ... | 9

**non\_zero\_digit**  $\rightarrow$  1 | ... | 9

**number**  $\rightarrow$  non\_zero\_digit digit\*

- Natural numbers
- Days of a 10-day week
- Colors
- ...

# Syntax vs. Semantics

---

**Structure** of  
code

**Meaning** of  
code

**Example:**

**Grammar to define a language:**    **Could mean**

**digit** → 0 | 1 | ... | 9

**non\_zero\_digit** → 1 | ... | 9

**number** → non\_zero\_digit digit\*

- Natural numbers
- Days of a 10-day week
- Colors
- ...

**Focus of this and next lecture**

# Syntax of Different PLs

---

**Common: Different syntax, same semantics**

*Java:*

```
if (foo > 100) {  
    ...  
}
```

*Bash:*

```
if [ $foo -gt 100 ]  
then  
    ...  
fi
```

# Syntax of Different PLs

---

**Common: Different syntax, same semantics**

*Java:*

```
if (foo > 100) {  
    ...  
}
```

*Bash:*

```
if [ $foo -gt 100 ]  
then  
    ...  
fi
```

**Sometimes: Same syntax, different semantics**

*Java:*

```
if ("abc" != 5) {  
    ...  
}
```

*JavaScript:*

# Syntax of Different PLs

---

Common: **Different syntax, same semantics**

*Java:*

```
if (foo > 100) {  
    ...  
}
```

*Bash:*

```
if [ $foo -gt 100 ]  
then  
    ...  
fi
```

Sometimes: **Same syntax, different semantics**

*Java:*

Type error at  
compile time

```
if ("abc" != 5) {  
    ...  
}
```

*JavaScript:*

Branch is  
executed

## 4 concepts to specify syntax

concatenation

alternation / choice

repetition ("Kleene closure")

recursion

} regular  
expression  
↓  
recognized  
by scanners

} context-free  
grammars (CFG)  
↓  
recognized  
by parsers

# Overview

---

- **Specifying syntax**
  - Regular expressions
  - Context-free grammars
- **Scanning**
- **Parsing**

# Tokens

---

## Basic **building blocks** of every PL

- Keywords, identifiers, constants, operators
- Think: "Words" of the language

## Example: C has more than 100 tokens

- Keywords, e.g., `double`, `if`, `return`, `struct`
- Identifiers, e.g., `my_var`, `printf`
- Literals, e.g., `6.022e23`, `'x'`
- Punctuators, e.g., `(`, `}`, `&&`

# Regular Expressions

---

- Used to **specify tokens**
- A regular expression is one of:
  - A **character**
  - The **empty string**  $\epsilon$
  - The **concatentation** of two regular expressions
  - Two regular expressions separated by **|**
    - Means a string generated by one or the other
  - A reg. expression followed by the **Kleene star**  $*$ 
    - Means zero or more repetitions

# Regular Expressions

---

- Used to **specify tokens**
- A regular expression is one of:
  - A **character**
  - The **empty string**  $\epsilon$
  - The **concatentation** of two regular expressions
  - Two regular expressions separated by **|**
    - Means a string generated by one or the other
  - A reg. expression followed by the **Kleene star**  $*$ 
    - Means zero or more repetitions

**No recursion!**



## Example

Numeric constants accepted by a calculator

number  $\rightarrow$  integer | real

integer  $\rightarrow$  digit digit\*

real  $\rightarrow$  integer exponent | decimal (exponent | E)

decimal  $\rightarrow$  digit\* (. digit | digit.) digit\*

exponent  $\rightarrow$  (e | E) (+ | - | E) integer

digit  $\rightarrow$  0 | 1 | ... | 9

# Quiz

---

**Which of the following strings is accepted by the regular expression *number*?**

- e4
- 007
- 0.123.45
- 7E-.517
- 5e-12
- 23+42+5E.0

# Quiz

---

Which of the following strings is accepted by the regular expression *number*?

- e4 ✗
- 007 ✓
- 0.123.45 ✗
- 7E-.517 ✗
- 5e-12 ✓
- 23+42+5E.0 ✗

More compact syntax for regular expressions

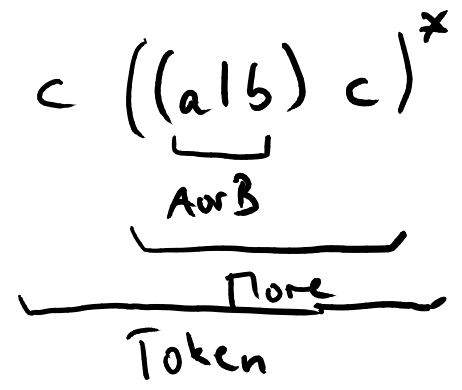
Language of tokens: { c, cac, cbc, cacac, cbcbc, cacbc, cbcac, ... }

Token → c More\*

More → AorB c

AorB → a|b

Shorter notation: Nest all into one



# Identifiers in Popular PLs

---

## Different PLs allow different identifiers

- Case-sensitive vs. case-insensitive
  - E.g., `f00`, `F00`, and `F00` are the same in Ada and Common Lisp, but not in Perl and C
- Letters and digits: Almost always allowed
- Underscore: Allowed in most languages

## In addition to syntax rules: Conventions

- E.g., Java: `ClassName`, `variableName`

# Identifiers in Popular PLs

---

## Different PLs allow different identifiers

- Case-sensitive vs. case-insensitive
  - E.g., `f00`, `F00`, and `F00` are the same in Ada and Common Lisp, but not in Perl and C
- Letters and digits: Almost always allowed
- Underscore: Allowed in most languages

## In addition to syntax rules: **Conventions**

- E.g., Java: `ClassName`, `variableName`

**Know the rules of the language you use!**

# White space in Popular PLs

---

## Free format **vs.** formatting as syntax

- Spaces and tabs sometimes matter
  - E.g., in Python
- Line breaks sometimes matter
  - E.g., to separate statements in JavaScript or Python

# Demo

---

**[demos/whitespace: show both stmts on one line; insert semi-colon; show not indenting print (after inverting condition)]**

# Overview

---

- **Specifying syntax**
  - Regular expressions
  - Context-free grammars ←
- **Scanning**
- **Parsing**

Are regular expressions enough?

Specifying arithmetic expressions

↳ E.g.,

$$\begin{array}{l} 5+7 \\ (5+7)+6 \\ ((5+7)+6)-23 \end{array}$$

Each  is an arithmetic expression

→ Want: Nesting

→ Need recursion

# Context-free Grammars

---

≈ **Regular expressions + Recursion**

**Example: Arithmetic expressions**

**expr** → **id** | **number** | **expr op expr** | **( expr )**

**op** → **+** | **-** | **\*** | **/**

# Context-free Grammars

---

≈ Regular expressions + Recursion

Example: Arithmetic expressions

**expr** → **id** | **number** | **expr op expr** | **( expr )**

**op** → **+** | **-** | **\*** | **/**

**Non-terminals**



# Context-free Grammars

---

≈ Regular expressions + Recursion

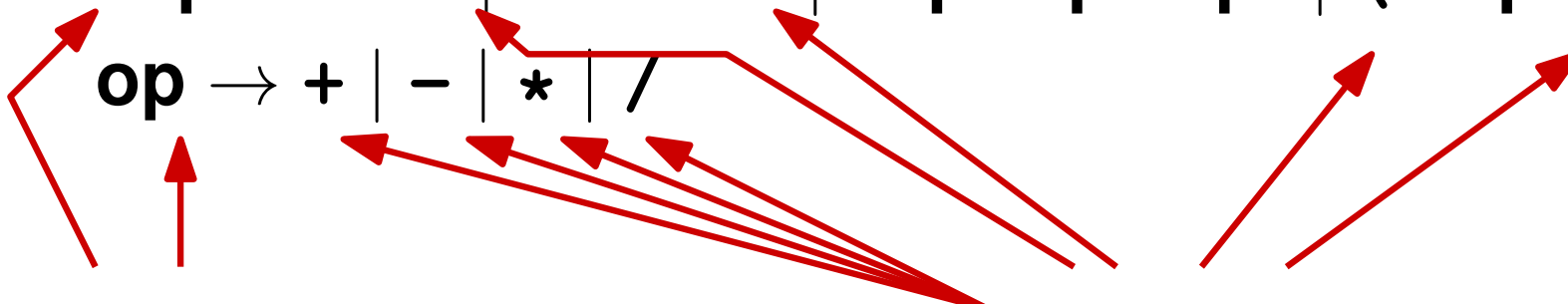
Example: Arithmetic expressions

**expr** → **id** | **number** | **expr op expr** | **( expr )**

**op** → **+** | **-** | **\*** | **/**

**Non-terminals**

**Terminals =  
tokens of the PL**



# Context-free Grammars

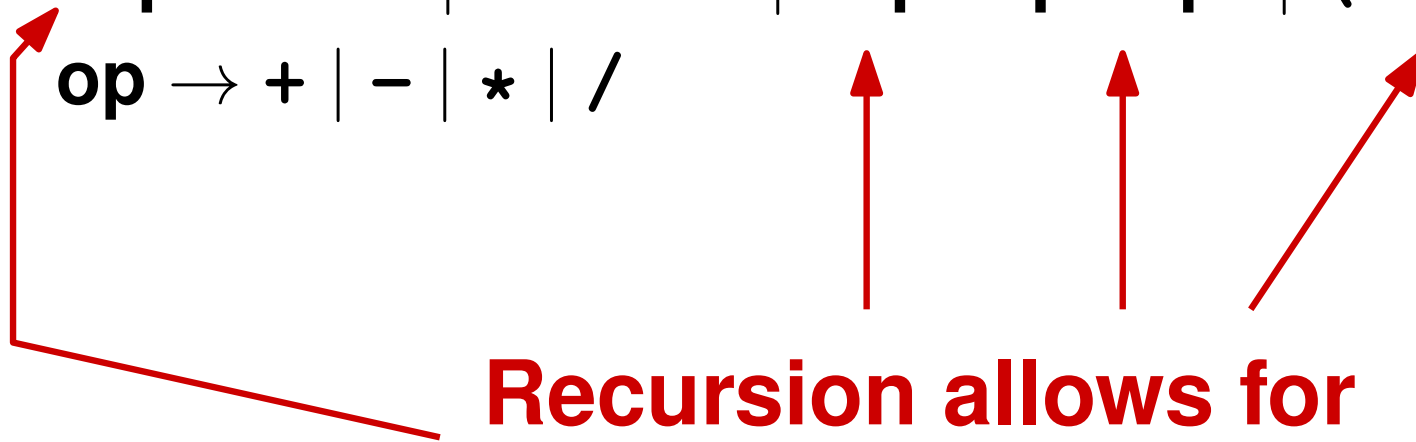
---

≈ Regular expressions + Recursion

Example: Arithmetic expressions

$\text{expr} \rightarrow \text{id} \mid \text{number} \mid \text{expr op expr} \mid (\text{expr})$

$\text{op} \rightarrow + \mid - \mid * \mid /$



**Recursion allows for nesting expressions**

## Definition of CFGs

$$G = (N, T, R, s)$$

$N$  .. finite set of non-terminals

$T$  .. finite set of terminals

= alphabet of the language

= (for PLs) tokens of the language

$R$  .. finite relation from  $N$  to  $(N \cup T)^*$

= production rules

$s$  .. start symbol

## Extension of basic definition

- Kleene star

↳ E.g.,  $\text{id\_list} \rightarrow \text{id}(\text{, id})^*$

means zero or more

↳ short-hand for

$\text{id\_list} \rightarrow \text{id}$

$\text{id\_list} \rightarrow \text{id\_list} \text{ , id}$

- Kleene plus

↳ Same, but one or more

- Vertical bar

↳ E.g.,  $\text{op} \rightarrow + | - | * | /$

↳ Short-hand for

$\text{op} \rightarrow +$

$\text{op} \rightarrow -$

⋮

# Derivations

---

## Create concrete strings from the grammar

- Begin with start symbol
- Repeat until no non-terminals remain:
  - Choose non-terminal and a production with this non-terminal on the left-hand side
  - Replace it with right-hand side of the production (choose one option if multiple options)

Example:  $\text{expr} \rightarrow \text{id} \mid \text{number} \mid (\text{expr}) \mid \text{expr op expr}$   
 $\text{op} \rightarrow + \mid - \mid * \mid /$

Derivation of  $\text{foo} * \text{x} + \text{bar}$

$\text{expr} \Rightarrow \text{expr op } \underline{\text{expr}}$   
 $\Rightarrow \text{expr } \underline{\text{op}} \text{ id}$   
 $\Rightarrow \underline{\text{expr}} + \text{id}$   
 $\Rightarrow \text{expr op } \underline{\text{expr}} + \text{id}$   
 $\Rightarrow \text{expr } \underline{\text{op}} \text{ id} + \text{id}$   
 $\Rightarrow \underline{\text{expr}} * \text{id} + \text{id}$   
 $\Rightarrow \text{id} * \text{id} + \text{id}$   
 $\text{foo}$ 
 $\text{x}$ 
 $\text{bar}$

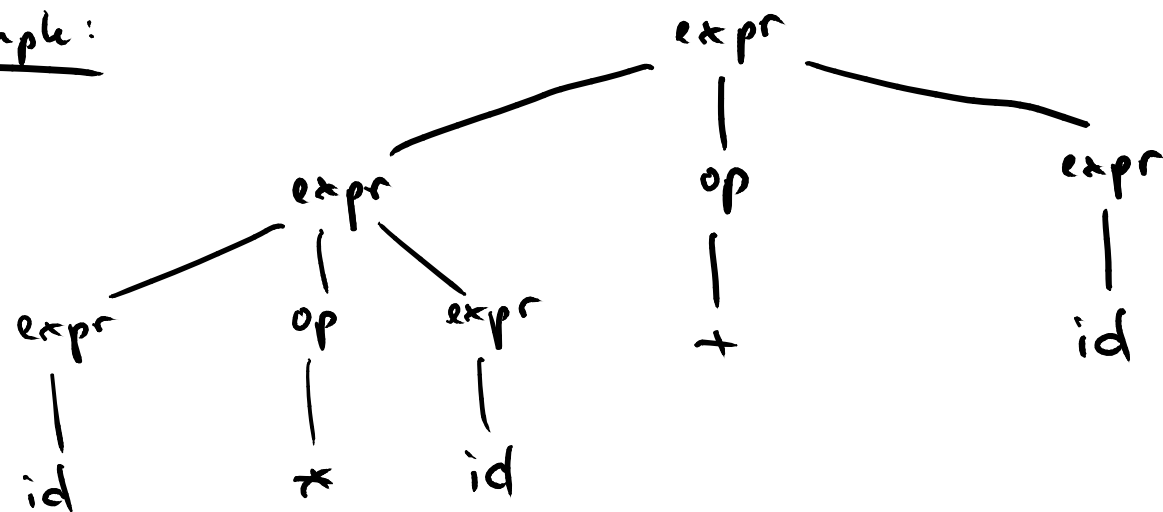
# Parse Trees

---

## Tree-structured representation of a derivation

- Root = Start symbol
- Leaf nodes = Tokens that result from derivation
- Intermediate nodes = Application of a production

Example:



# Not All Grammars are Equal

---

Each language has **infinitely many grammars**

Some grammars are **ambiguous**

- A single string may have multiple derivations
- Unambiguous grammars facilitate parsing

Grammar should **reflect the internal structure** of the PL

- E.g., associativity and precedence of operators

# Example: Revised Grammar

---

**A better version of the grammar of arithmetic expressions:**

**expr  $\rightarrow$  term | expr add\_op term**

**term  $\rightarrow$  factor | term mult\_op factor**

**factor  $\rightarrow$  id | number | - factor | ( expr )**

**add\_op  $\rightarrow$  + | -**

**mult\_op  $\rightarrow$  \* | /**

# Quiz: Context-free Grammars

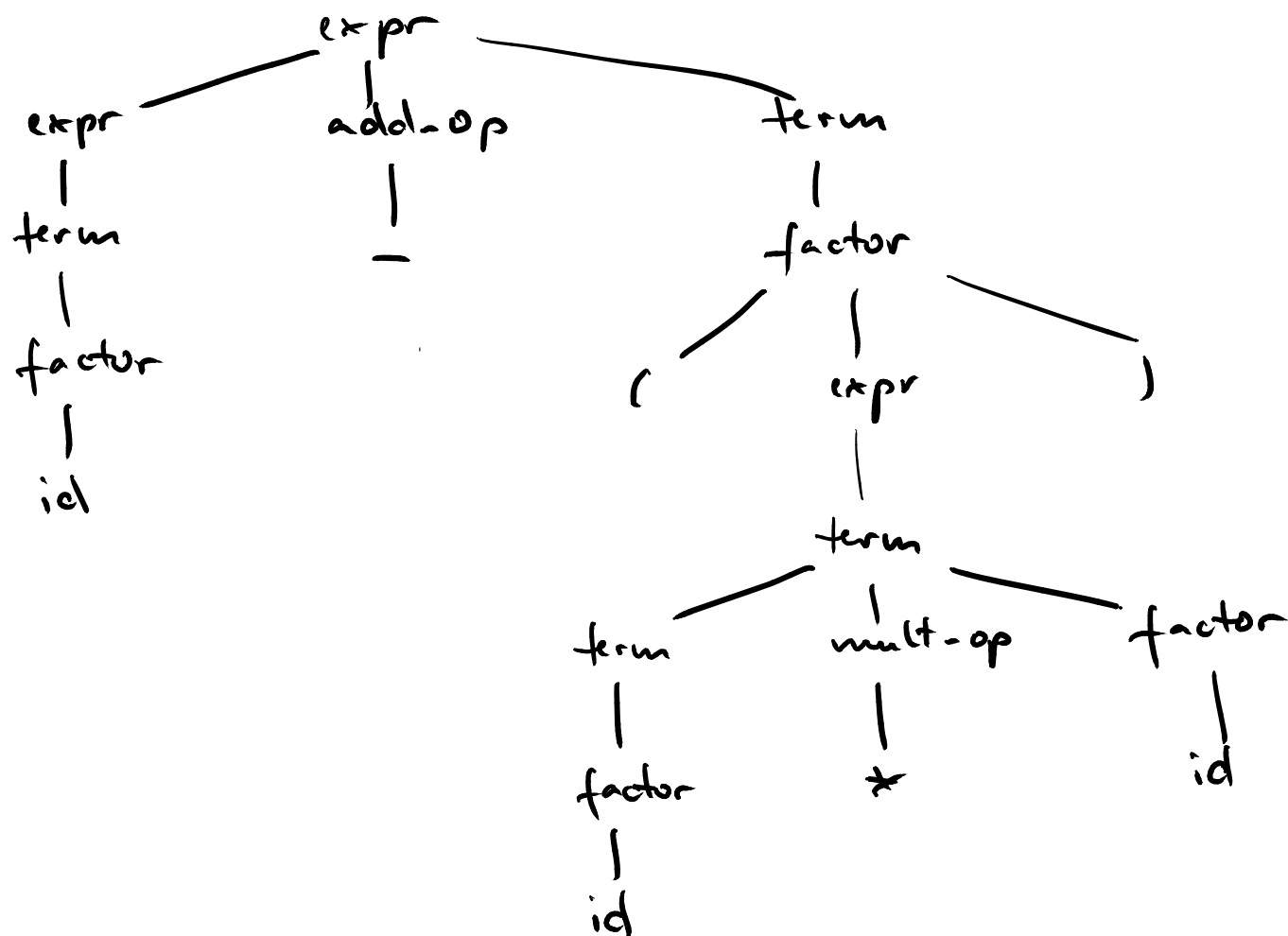
---

**Draw the parse tree of**

**foo - (bar \* bar)**


**with the revised grammar. How many nodes and edges does the tree have?**

foo - (bar \* bar)



# Overview

---

- **Specifying syntax**
  - Regular expressions
  - Context-free grammars
- **Scanning** 
- **Parsing**

# Implementing a Scanner

---

## General idea

- Read **one character at a time**
- Whenever a **full token** is recognized, return it
- When no token can be recognized, report an **error**
- Sometimes, need to **look multiple characters ahead** to determine next token

# Option 1: Ad-hoc Scanners

---

- **Manually** implemented
- Handle common tokens first
- Used in many **production compilers**
  - Compact code
  - Efficient scanning

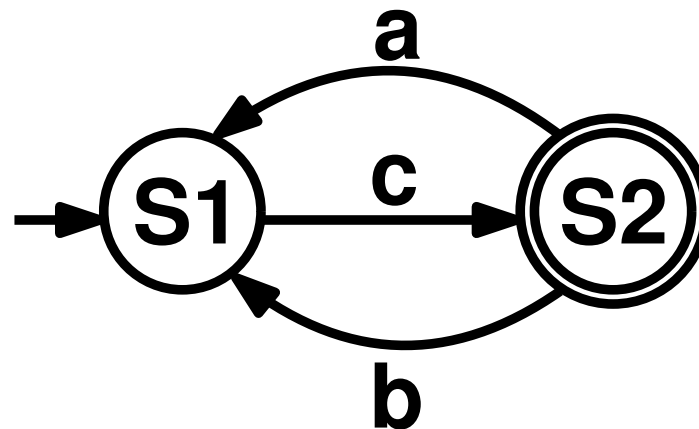
# Option 2: Finite Automata

---

- Each token specified by a regular expression
- **Finite automata = Recognizers of regular expressions**

**Example:**

$c ((a | b) c)^*$



# Definition: DFA

---

## Deterministic finite automaton (DFA):

$(Q, \Sigma, \delta, q_0, F)$

- Finite set  $Q$  of states
- Finite set  $\Sigma$  of input symbols
- Transition function  $\delta : Q \times \Sigma \rightarrow Q$
- Start state  $q_0$
- Set of accept states  $F \subseteq Q$

# DFA versus NFA

---

- **Deterministic finite automaton (DFA):**

- At most one outgoing transition for each input symbol
- No  $\epsilon$  transitions (empty word)

- **Non-deterministic finite automaton (NFA)**

- Multiple outgoing transitions for same character
- May have  $\epsilon$  transitions

# From Reg. Expr. to DFA

---

- **Regular expression to NFA**
- **NFA to DFA**
  - To avoid exploring multiple possible next states during scanning
- **DFA to minimal DFA**
  - Simplifies a DFA-based scanner
  - Remove unreachable and non-distinguishable states

See course on theoretical computer science or Chapter 2 of “Programming Language Pragmatics” for details