Programming Languages in Software Engineering

Lecture 2

Komi Golova (she/her)
komi.golov@jetbrains.com

Constructor University Bremen

Announcements

Don’t forget the homework!

- Due Sunday evening
- Send a PR to https://github.com/jesyspa/cub-plsd-2025-student

Parsing

Parsing is the process of turning unstructured data into structured data.

Parsing tools exist, and are easy to use (once you know how to). Even if you never need to do PL, being able to parse data is useful.

Typically, parsing happens in two steps:

1. Lexing: turn the input string into a list of meaningful chunks.
2. Parsing: turn the list into a tree.

Basics

Parsing turns a string into some more complicated structure.

The string is taken from a fixed *alphabet* (e.g. ASCII, Unicode, bytes, lexer tokens). We call such a string a *word*.

Words and elements of the alphabet are called *terminals*.

Helper variables are called *non-terminals*.

Regular expressions

Regular expressions are a succinct way to describe a set of strings.

Regular expressions

Regular expressions are a succinct way to describe a set of strings.

Rules:

- ε matches the empty string
- $\emptyset$ matches nothing
- `a` matches the character “a”
- `pq` matches the pattern `p` and then the pattern `q`
- `p|q` matches the pattern `p` or the pattern `q`
- `p*` matches the pattern `p` zero or more times

Regular expressions

Regular expressions are a succinct way to describe a set of strings.

Rules:

- ε matches the empty string
- $\emptyset$ matches nothing
- `a` matches the character “a”
- `pq` matches the pattern `p` and then the pattern `q`
- `p|q` matches the pattern `p` or the pattern `q`
- `p*` matches the pattern `p` zero or more times

Regular expressions can (in theory) be parsed in linear time and space.

Regular expressions in practice

In practice, most engines will provide more features:

- Shorthands: `.`, `[a-z]`, `p+`, `p?`, `p{n}`...
- Positional matches: `^`, `$`, `\b`...
- Grouping support: `(p)` lets you extract the string `p` matched
- Back-references: `\1` to refer to first group

Regular expressions in practice

In practice, most engines will provide more features:

- Shorthands: `.`, `[a-z]`, `p+`, `p?`, `p{n}`...
- Positional matches: `^`, `$`, `\b`...
- Grouping support: `(p)` lets you extract the string `p` matched
- Back-references: `\1` to refer to first group

In practice, regular expressions are not regular.

Runtime may be exponential even for simple cases:

`(a+)*` matching `aaa...`

This will try *all* ways of splitting the input string!

Regular expressions are pain

Can I validate an email address with regex?

Regular expressions are pain

Can I validate an email address with regex? Sure!

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}
~-]+)*|"(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-
\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:(?:[a-z0-9](?:[a-
z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\[(?:
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|
2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:(?:[\x01-
\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-
\x09\x0b\x0c\x0e-\x7f])+)\])
```

Just because you can, doesn't mean you should.

Context-free grammars

Regular expressions cannot parse recursive structures.

For example, S-expressions: `(lambda (x) (+ x 1))`

Context-free grammars

Regular expressions cannot parse recursive structures.

For example, S-expressions: `(lambda (x) (+ x 1))`

For that, we want more complicated grammars:

$$
\begin{aligned}
E &\to \text{Name} \\
E &\to \text{Number} \\
E &\to (E') \\
E' &\to EE' \\
E' &\to \varepsilon
\end{aligned}
$$

Context-free grammars

Regular expressions cannot parse recursive structures.

For example, S-expressions: `(lambda (x) (+ x 1))`

For that, we want more complicated grammars:

$$
\begin{aligned}
E &\to \text{Name} \\
E &\to \text{Number} \\
E &\to (E') \\
E' &\to EE' \\
E' &\to \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
E &\to \text{Name} \mid \text{Number} \mid (E') \\
E' &\to EE' \mid \varepsilon
\end{aligned}
$$

Context-free grammars

Regular expressions cannot parse recursive structures.

For example, S-expressions: `(lambda (x) (+ x 1))`

For that, we want more complicated grammars:

$$
\begin{aligned}
E &\to \text{Name} \\
E &\to \text{Number} \\
E &\to (E') \\
E' &\to EE' \\
E' &\to \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
E &\to \text{Name} \mid \text{Number} \mid (E') \\
E' &\to EE' \mid \varepsilon
\end{aligned}
$$

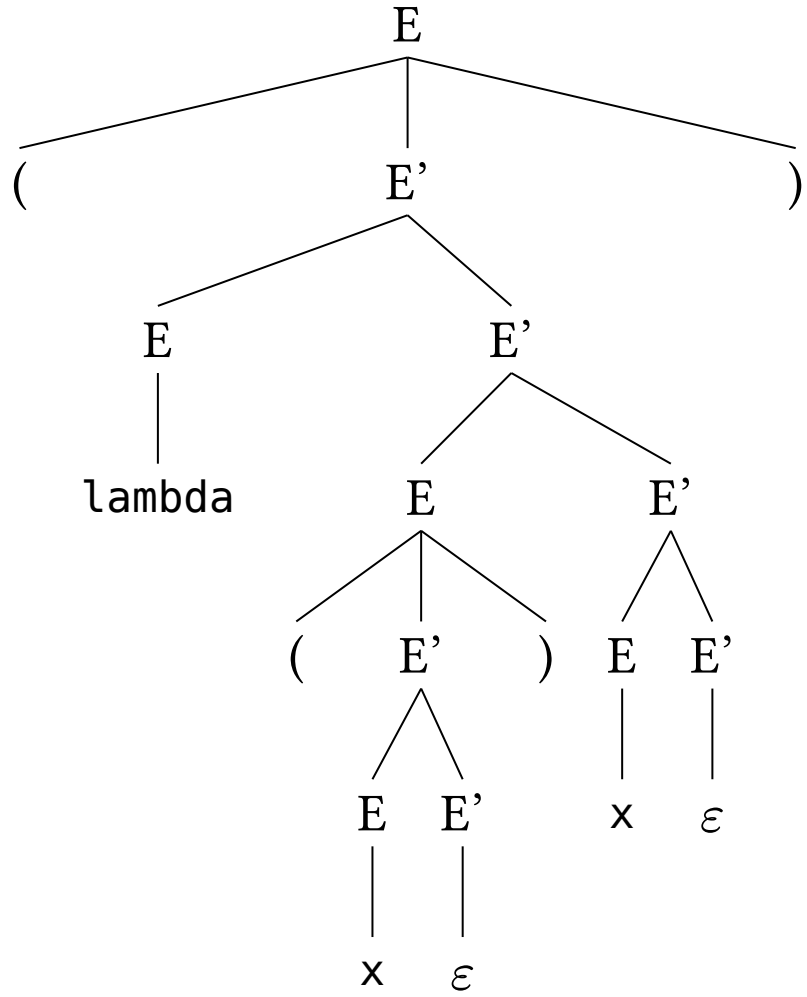
- E, E' are non-terminals
- Name, Number, (, and) are terminals
  - Name and Number are *tokens*
- ε represents the empty string
- Order of rules does not matter

Context-free grammars (example)

$$E \to \text{Name} \mid \text{Number} \mid (E')$$

$$E' \to EE' \mid \varepsilon$$

```
(lambda (x) x)
```



Separate lexers

We used Name and Number as terminals, but our input is a string.

We can still do this by saying we lex our input before parsing.

Separate lexers

We used Name and Number as terminals, but our input is a string.
We can still do this by saying we lex our input before parsing.

Pure CFG:

$$
\begin{aligned}
E &\to \text{Name} \mid \text{Number} \mid (E') \\
E' &\to EE' \mid \varepsilon \\
\text{Name} &\to \text{Letter NameTail} \\
\text{NameTail} &\to \text{AlphaNum NameTail} \mid \varepsilon \\
\text{AlphaNum} &\to \text{Letter} \mid \text{Digit} \\
\text{Number} &\to \text{Digit NumberTail} \\
\text{NumberTail} &\to \text{Digit NumberTail} \mid \varepsilon \\
\text{Letter} &\to \text{a} \mid \ldots \mid \text{z} \\
\text{Digit} &\to 0 \mid \ldots \mid 9
\end{aligned}
$$

Separate lexers

We used Name and Number as terminals, but our input is a string.
We can still do this by saying we lex our input before parsing.

Pure CFG:

$$
\begin{aligned}
E &\to \text{Name} \mid \text{Number} \mid (E') \\
E' &\to EE' \mid \varepsilon \\
\text{Name} &\to \text{Letter NameTail} \\
\text{NameTail} &\to \text{AlphaNum NameTail} \mid \varepsilon \\
\text{AlphaNum} &\to \text{Letter} \mid \text{Digit} \\
\text{Number} &\to \text{Digit NumberTail} \\
\text{NumberTail} &\to \text{Digit NumberTail} \mid \varepsilon \\
\text{Letter} &\to \text{a} \mid \ldots \mid \text{z} \\
\text{Digit} &\to 0 \mid \ldots \mid 9
\end{aligned}
$$

RegEx lexer + CFG:

$$
\begin{aligned}
E &\to \text{Name} \mid \text{Number} \mid (E') \\
E' &\to EE' \mid \varepsilon \\
\text{Name} &\sim\!= \text{"[a-zA-Z][a-zA-Z0-9]*"} \\
\text{Number} &\sim\!= \text{"0|[1-9][0-9]*"}
\end{aligned}
$$

Separate lexers

We used Name and Number as terminals, but our input is a string.
We can still do this by saying we lex our input before parsing.

Pure CFG:

$$
\begin{aligned}
E &\to \text{Name} \mid \text{Number} \mid (E') \\
E' &\to EE' \mid \varepsilon \\
\text{Name} &\to \text{Letter NameTail} \\
\text{NameTail} &\to \text{AlphaNum NameTail} \mid \varepsilon \\
\text{AlphaNum} &\to \text{Letter} \mid \text{Digit} \\
\text{Number} &\to \text{Digit NumberTail} \\
\text{NumberTail} &\to \text{Digit NumberTail} \mid \varepsilon \\
\text{Letter} &\to \text{a} \mid \ldots \mid \text{z} \\
\text{Digit} &\to 0 \mid \ldots \mid 9
\end{aligned}
$$

RegEx lexer + CFG:

$$
\begin{aligned}
E &\to \text{Name} \mid \text{Number} \mid (E') \\
E' &\to EE' \mid \varepsilon \\
\text{Name} &\sim\!= \text{"[a-zA-Z][a-zA-Z0-9]*"} \\
\text{Number} &\sim\!= \text{"0|[1-9][0-9]*"}
\end{aligned}
$$

In CFG-based systems, using a separate lexer is very common.

Context-free grammars (problem)

Consider this context-free grammar for expressions:

$$E \to E + E \mid E - E$$

$$E \to \text{Number}$$

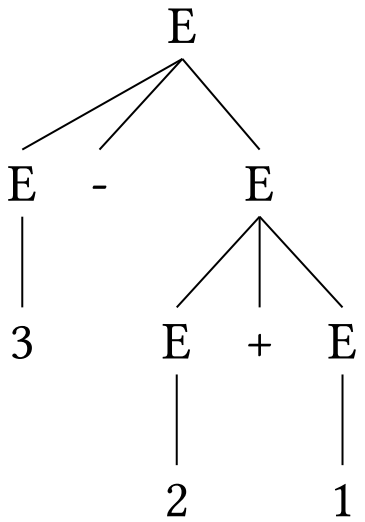
Context-free grammars (problem)

Consider this context-free grammar for expressions:

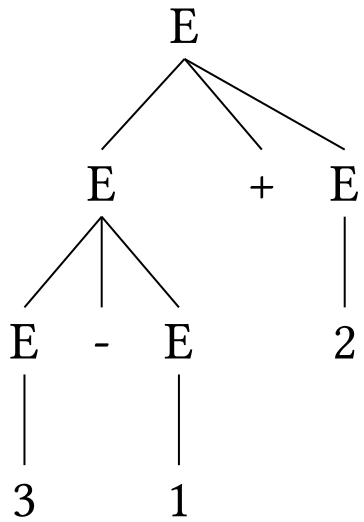
$$E \to E + E \mid E - E$$

$$E \to \text{Number}$$

Parsing $3 - 2 + 1$ gives:



$= 2$



$= 4$

Context-free grammars (problem)

Consider this context-free grammar for expressions:

$$E \to E + E \mid E - E$$

$$E \to \text{Number}$$

We can fix this in this case:

$$E \to E + \text{Number} \mid E - \text{Number}$$

$$E \to \text{Number}$$

Practical issues with ambiguity

Consider the following program and grammar:

```
if (a)
  if (b)
    print("yes");
else
  print("no");
```

$$S \rightarrow \text{if } (C)\ S\ E$$

$$S \rightarrow \text{print } (\text{String});$$

$$E \rightarrow \varepsilon \mid \text{else } S$$

$$C \rightarrow \text{Variable}$$

Practical issues with ambiguity

Consider the following program and grammar:

```
if (a)
  if (b)
    print("yes");
else
  print("no");
```

$$S \rightarrow \text{if } (C)\ S\ E$$

$$S \rightarrow \text{print } (\text{String});$$

$$E \rightarrow \varepsilon \mid \text{else } S$$

$$C \rightarrow \text{Variable}$$

With a CFG, the standard approach is to parse and then disambiguate.

Parsing expression grammars

Regexes are succinct but limited. CFGs are powerful but fiddly.

Parsing expression grammars are a middle ground.

Parsing expression grammars

Regexes are succinct but limited. CFGs are powerful but fiddly.

Parsing expression grammars are a middle ground.

PEG for S-expressions:

```
SExpr <- OPEN SExpr* CLOSE / Literal / Identifier
OPEN <- '(' Spacing
CLOSE <- ')' Spacing
Literal <- '-'? [1-9] [0-9]* Spacing
Identifier <- IdentStart IdentCont* Spacing
IdentStart <- [a-zA-Z_-+*]
IdentCont <- IdentStart / [0-9]
Spacing <- [ \n]*
```

Comparison

Like CFGs:

- Set of rules (`S <- a b c`)

Like regexes:

- Support for `[a-z]`, `p*`, `p?`...
- Linear-time parsing

Comparison

Like CFGs:

- Set of rules (`S <- a b c`)

But now:

- Choice is left-biased:
  - `"a" / "ab"` always matches `a`

Like regexes:

- Support for `[a-z]`, `p*`, `p?`...
- Linear-time parsing

Comparison

Like CFGs:

- Set of rules (`S <- a b c`)

Like regexes:

- Support for `[a-z]`, `p*`, `p?`...
- Linear-time parsing

But now:

- Choice is left-biased:
  - `"a" / "ab"` always matches `a`
- Support for lookahead:
  - `&p` matches `p` and then backtracks on success
  - `!p` matches `p` and then backtracks on failure

Email

```
name_chunk <- [a-z0-9!#$%&'*+/=?^_`{|}~-]+
special <- [\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]
         / '\\' [\x01-\x09\x0b\x0c\x0e-\x7f])
quoted_name <- '"' special* '"'
name <- name_chunk ('.' name_chunk)* / quoted_name
ident_end <- [a-z0-9-]* [a-z0-9]
ident <- [a-z0-9] ident_end?
ident_host <- ident ('.' ident)+
byte <- '25' [0-5] / '2' [0-4] [0-9] / [01]? [0-9] [0-9]?
end <- byte / ident_end ':' special+
host <- ident_host / '[' byte '.' byte '.' byte '.' end ']'
email <- name '@' host
```

Lookahead

Regex has built-in support for `[^\n]`: match anything except a newline.

In a PEG we can simulate this:

```
NotNewline <- (!'\n' .)
```

Lookahead

Regex has built-in support for `[^\n]`: match anything except a newline.
In a PEG we can simulate this:

```
NotNewline <- (!'\n' .)
```

Application, C++-style comments:

```
Comment <- "/*" (!"*/" .)* "*/"
```

Parsing in Practice

Libraries for parsing are available in most languages. Two styles:

- *Parser generators* build a parser from a grammar.
- *Parser combinators* let you express the grammar in your language.

Parser generators

A *parser generator* takes a separate grammar definition and generates a parser for you. Many options exist: lex/yacc, ANTLR, ...

Example from the ANTLR docs:

Parser generators

A *parser generator* takes a separate grammar definition and generates a parser for you. Many options exist: lex/yacc, ANTLR, ...

Example from the ANTLR docs:

```
grammar Expr;
prog: expr EOF ;
expr: expr ('*'|'/') expr
    | expr ('+'|'-') expr
    | INT
    | '(' expr ')'
    ;
INT     : [0-9]+ ;
```

Parser generators

A *parser generator* takes a separate grammar definition and generates a parser for you. Many options exist: lex/yacc, ANTLR, ...

Example from the ANTLR docs:

```
grammar Expr;
prog: expr EOF ;
expr: expr ('*'|'/') expr
    | expr ('+'|'-') expr
    | INT
    | '(' expr ')'
    ;
INT     : [0-9]+ ;
```

Features:

- Constructs a parser as part of your build.
- Automatic disambiguation of left-recursive rules.
- Support for visiting the tree and listening for parser events.

Parser combinators

Parser combinators are a way to build your parser in the language you're parsing.

Example from the PyParsing docs:

```
expression = infix_notation(
    integer | variable,
    [
        ("-", 1, OpAssoc.RIGHT),
        (one_of("* /"), 2, OpAssoc.LEFT),
        (one_of("+ -"), 2, OpAssoc.LEFT),
    ]
)
```

Applicative parser combinators

Let's say we want to parse and evaluate expresions like `(5 + 3) + 2`.

Applicative parser combinators

Let's say we want to parse and evaluate expresions like `(5 + 3) + 2`.

Attempt #1:

```
expr = parse_rule { rec ->
  +regex("-?\\d+")
  +(rec + char('+') + rec)
  +(char('(') + rec + char(')'))
}
```

But now we need write a separate evaluator...

Applicative parser combinators

Let's say we want to parse and evaluate expresions like `(5 + 3) + 2`.

Attempt #2:

```
expr = parse_rule { rec ->
  +regex("-?\\d+").map { num -> parseInt(num) }
  +(rec + char('+') + rec).map { (x, _, y) -> x + y }
  +(char('(') + rec + char(')')).map { (_, e, _) -> e }
}

>>> parse(expr, "(5+3)+2")
10
```

Monadic parser combinators

Let's say we're training a classifier model and have the following binary format:

```
vector_size: 2 bytes
num_positive_examples: 4 bytes
num_negative_examples: 4 bytes
pos_examples: num_positive_examples times vector_size bytes
neg_examples: num_negative_examples times vector_size bytes
```

What problem are we going to run into using a parser for this?

Monadic parser combinators

Let's say we're training a classifier model and have the following binary format:

```
vector_size: 2 bytes
num_positive_examples: 4 bytes
num_negative_examples: 4 bytes
pos_examples: num_positive_examples times vector_size bytes
neg_examples: num_negative_examples times vector_size bytes
```

What problem are we going to run into using a parser for this?

Grammars do not allow us to use the value of `vector_size` to parse the rest of the file.

Monadic parser combinators

Format: `s: 2 | npos: 4 | nneg: 4 | pos: s*npos | neg: s*nneg`

```
file = parse {
  bytes(2).map { asInt(it) }.andThen { size ->
    bytes(4).map { asInt(it) }.andThen { npos ->
      bytes(4).map { asInt(it) }.andThen { nneg ->
        (bytes(size*npos) + bytes(size*nneg)).map {
          // processing
        }
      }
    }
  }
}
```

Parser combinator comparison

Applicative parser combinators:

- Structure of the parser is fixed.
- Like a parser generator, but executed at runtime.
- Can specify logic for processing results.

Monadic parser combinators:

- Structure of the parser determined during parsing.
- More powerful than CFG or PEG.
- Hard to optimise.

Practical guidelines

RegEx:

- Easy to start
- Hard to maintain
- Limited power
- Good for lexers

CFG:

- May require a lexer
- Can be fiddly
  - Ambiguity
  - Restrictions for performance

PEG:

- Strong performance guarantees
- Parsing behaviour can be surprising

Practical guidelines

RegEx:

- Easy to start
- Hard to maintain
- Limited power
- Good for lexers

CFG:

- May require a lexer
- Can be fiddly
  - Ambiguity
  - Restrictions for performance

PEG:

- Strong performance guarantees
- Parsing behaviour can be surprising

Parser generators:

- May have better errors
- May be more restrictive
- May perform better

Parser combinators:

- Easier to integrate
- Require language support