Formal languages

Principles of Programming Languages

Mark Armstrong

Fall 2020

1 Preamble

1.1 TODO Notable references

:TODO:

2 Introduction

This section introduces the mathematical tools we will use in the discussion of programming languages as a *formal* language.

Several small formal languages (not full programming languages) are used as examples of the use of these tools.

3 Formal languages

A language over an *alphabet* (set of symbols) Σ is a subset of Σ^*. The elements of a language are called *sentences* (or *strings* or sometimes *words*).

A *formal* language is one for which we have a mathematical tool for either

- *generating* (or *deriving*) all sentences of the language, or equivalently,
- *recognising* (or *accepting*) only sentences of the language.

Examples of such mathematical tools include

- regular expressions,

- automata, and
- grammars.

3.1 The usefulness of formal languages

Formal languages, unlike *natural* languages, are well-suited for comprehension by computers.

- Computers require unambiguous steps to follow.
- Hence, all programming languages are formal languages.

In particular, in most cases:

- The sets of keywords, names, etc. form several *regular languages*, and so can be recognised[1] by regular expressions.
- The set of valid (in terms of form) programs forms a *context-free* language, and so can be recognised by a (context-free) grammar.

3.2 Strings

Recall that given a set Σ, the set of strings over Σ, written Σ^*, is the set of all finite sequences of elements of Σ.

In particular, the sequence of length zero we denote by ε. Note that some other sources use λ for this purpose.

For example, for $\Sigma = \{a, b, c\}$,

$$\Sigma^* = \{\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, ...\}.$$

Given an element $e \in \Sigma$, we write

- e^n for the string consisting of n occurrences of e, and
- e^* for the set $\{n \in \mathbb{N} \mid e^n\}$.

[1] The computer must recognise (accept) valid programs. Conveniently, both regular expressions and grammars can also be viewed as generators (derivers), which is a useful point of view for humans reading the grammar/expression.

4 Describing the *syntax* of formal languages

Here, we will

- briefly review regular expressions and grammars as they are presented in formal language theory, and then
- introduce more practical syntax for each which is used in practice.

In both cases, the additional syntax only adds to the *practical expressiveness* of the tool.

- It does not change the *theoretical expressiveness* of the tool.
  - The same set of languages can be described, but many languages can be described "more easily".
- We will present brief arguments to this effect by showing how to translate from the new syntax to the restricted syntax.

4.1 Regular expressions as in formal language theory

:TODO:

4.2 Additional operators for more expression regexps

:TODO:

4.3 Regular expression examples

:TODO:

4.4 Grammars as in formal language theory

Formally, a context-free grammar is a 4-tuple

$$\langle N, \Sigma, P, S \rangle$$

where

- N is a finite set of *non-terminal* symbols (sometimes called variables),
- Σ is the underlying alphabet, also called the *terminals* of the grammar,
- N and Σ must be distinct,

- P is a set of *productions* i.e., a binary relation between N and

$$(N \cup \Sigma)^*$$

,

  - In other words, a multi-valued function from nonterminals to strings of non-terminals and terminals,

- S is a distinguished element of N, called the *starting nonterminal*.

Given

$$(A, \alpha) \in P$$

, we write

$$A \longrightarrow \alpha$$

and read it as "A produces α" or "A expands to α".

Given a number of productions $(A, \alpha_1) \in P$, $(A, \alpha_2) \in P$, ..., $(A, \alpha_m) \in P$, we write $A \longrightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_m$ as a shorthand.

4.5 Conventions for grammars

Writing the 4-tuple each time we produce a grammar is tedious.

For this reason, we adopt the following conventions in order to allow us to omit the 4-tuple.

1. We write *only* the list of production, using *BNF* (discussed shortly).
2. The set N is taken to be the set of all symbols appearing to the left of a list of productions.
   - Note that this requires each nonterminal have at least one production[fn:nonterminal-w/o-production].
3. The set Σ is usually understood by the context in which we are defining the grammer.
   - For our purposes, it will usually be the set of all ASCII symbols.
4. The starting nonterminal S is understood to be either
   (a) the nonterminal whose name matches that of the grammar we are defining (it may be uncapitalised or abbreviated),
   (b) otherwise, the non-terminal named S, or

(c) otherwise, the nonterminal to the left of the first production in the list,.

As a rule of thumb, we try to write grammars "top down", so that most nonterminals appearing to the right of a production have their rules listed below that production.

[fn:nonterminal-w/o-production] A nonterminal without productions has no practical use in any case; it only serves to making parsing "get stuck".

4.6 A simple example grammar

$$\begin{aligned} \texttt{A} &\longrightarrow \texttt{aAa} \mid \texttt{B} \\ \texttt{B} &\longrightarrow \texttt{bBb} \mid \texttt{C} \\ \texttt{C} &\longrightarrow \texttt{cCc} \mid \varepsilon \end{aligned}$$

This produces the language of strings of the form $\texttt{a}^i\texttt{b}^j\texttt{c}^k\texttt{c}^k\texttt{b}^j\texttt{a}^i$.

4.7 Exercise – reading grammars

What languages do the following grammars produce?

$$\begin{aligned} \texttt{A} &\longrightarrow \texttt{B} \mid \texttt{C} \\ \texttt{B} &\longrightarrow \texttt{aaB} \mid \varepsilon \\ \texttt{C} &\longrightarrow \texttt{aaaC} \mid \varepsilon \end{aligned}$$

$$\begin{aligned} \texttt{A} &\longrightarrow \texttt{aB} \mid \texttt{B} \mid \varepsilon \\ \texttt{B} &\longrightarrow \texttt{bC} \mid \texttt{C} \\ \texttt{C} &\longrightarrow \texttt{cA} \mid \texttt{A} \end{aligned}$$

$$\begin{aligned} \texttt{A} &\longrightarrow \texttt{aA} \mid \texttt{B} \\ \texttt{B} &\longrightarrow \texttt{bB} \end{aligned}$$

What's the tricky part with this one?

Extra exercise: can you simplify any of them? If you believe so, be careful that your simplification accepts the same string!

4.8 Parse trees

We have discussed the facts that a grammar can

- generate strings or
- recognise/accept strings.

Then for a grammar G we might think of functions

- $generate^G : \mathbb{N} \to \Sigma^*$
  - with the intention that $generate^G n$ generates the n^{th} string in the grammar's language is lexicographic order
- $recognise^G : \Sigma^* \to Bool$

That is, we have two functions, which output a `String` or a `Bool` respectively.

But there is a useful byproduct which may be obtained during during either process: a *parse tree*.

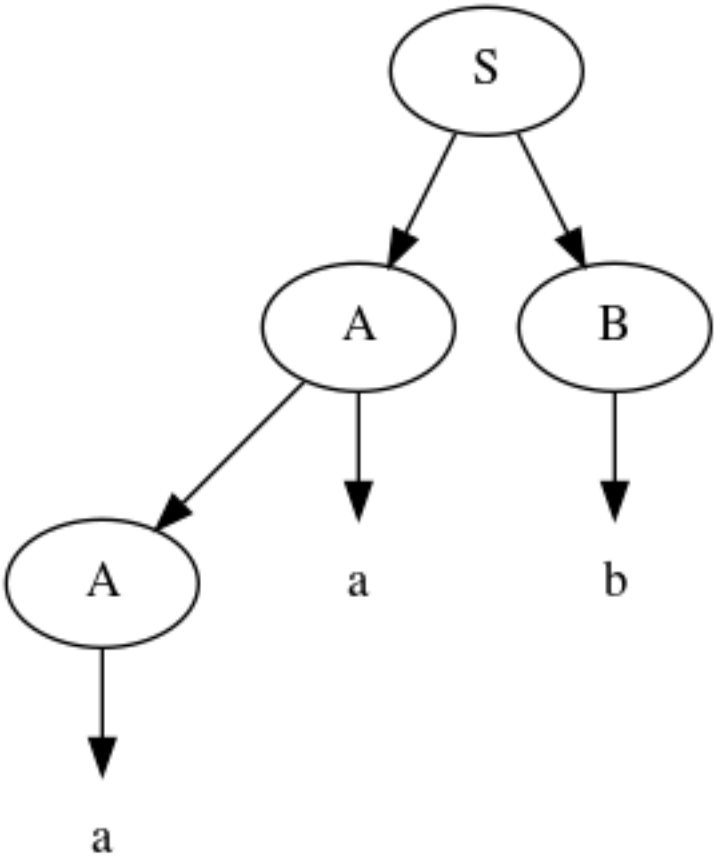
A parse tree's

- nodes (which have children) are labelled by a nonterminal of the grammar,
- leaves (which do not have children) are labelled by a terminal of the grammar or a nonterminal which may produce ε, and
- if a node is labelled by a nonterminal `A`, the children of that node must correspond (in order from left to right) the nonterminals appearing in a production of `A`.

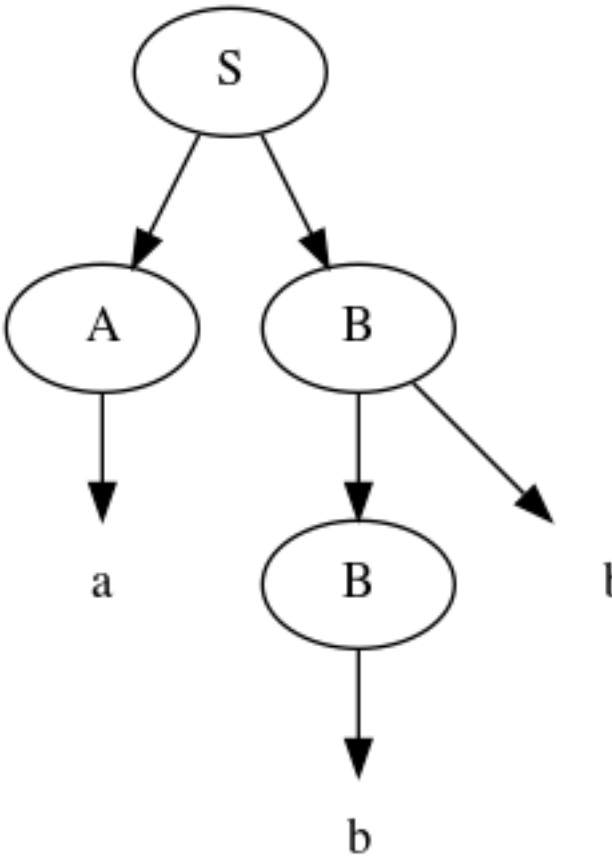
For example, if we consider the grammar

`S` $\longrightarrow$ `AB`
`A` $\longrightarrow$ `aA` | ε
`B` $\longrightarrow$ `Bb` | `b`

the following are valid parse trees.



First, for the string `aab`.



Similarly, we have the following parse tree for `abb`.

Exercise: provide a parse tree for the string `aaa` using this grammar. Is there a valid parse tree for the string `bbb`?

Exercise: if we add a production $\texttt{A} \longrightarrow \texttt{a}$ to our example grammar, can you provide a different parse tree (or multiple different parse trees) for `aaa`?

4.9 Backus-Naur form (BNF)

Up until now, we have used the form

$$\begin{array}{l} \texttt{N}_1 \longrightarrow \texttt{P}_1 \ \texttt{|} \ \texttt{P}_2 \ \texttt{|} \ \ldots \\ \quad \vdots \end{array}$$

for our production lists.

Commonly in the study of programming languages, an alternative syntax called *Backus-Naur* form (BNF) is used.

- Named for two members of the Algol[2] design committee, who created the first formal definition for a programming language, namely Algol.

In Backus-Naur form,

- all nonterminals names are delimited by angle brackets[3], ⟨⟩,
- the ⟶ is replaced by `::=`,
- additional whitespace is permitted on the right side of a production between terminals and nonterminals, without changing the meaning of the production
  - So ⟨`A`⟩ `::= a a` ⟨`A`⟩ is treated the same as ⟨`A`⟩ `::= aa`⟨`A`⟩.

4.10 Extended Backus-Naur form (EBNF)

We also extend our grammar notation to include several several additional operators.

- (Square) brackets, `[]`, surrounding a string indicate that string may or may not be included in a production.
  - I.e., they make part of a production optional.
  - ⟨`A`⟩ `::=` α_1 `[` α_2 `]` α_3 ≈ ⟨`A`⟩ `::=` α_1 α_2 α_3 `|` α_1 α_3.
- (Curly) braces, `{}`, surrounding a string indicate that string may be repeated any number of times, including zero.
  - ⟨`A`⟩ `::=` α_1 `{` α_2 `}` α_3 ≈ ⟨`A`⟩ `::=` α_1 ⟨`A`⟩ α_3 together with ⟨`A`⟩ `::=` α_2 ⟨`A`⟩ `|` ε.
- Parentheses, `()`, may group parts of a string.
- The "alternative" pipe, `|`, may be used *inside* of productions, to indicate alternatives inside a set of brackets, braces or parentheses.
  - ⟨`A`⟩ `::=` α_1 `(`α_2 `|` α_3`)` α_4 ≈ ⟨`A`⟩ `::=` α_1 α_2 α_4 `|` α_1 α_3 α_4.

[2] Algol was a contemporary of Fortran, Lisp, and Cobol, together the oldest languages still in (fairly) common use today. Algol is not in common use, but it was the most influential on modern programming language syntax, introducing concepts such as the block.

[3] In notes and assignments, I use unicode angle brackets. Many other sources use the less-than and greater-than symbols, as they are avaiable in ASCII.

- Where necessary, terminals may be single or double quoted, such as to indicate a whitespace character, pipe or quote.
  - ⟨ebnf-prod-list⟩ ::= ⟨string⟩ | ⟨string⟩ ⟨opt-ws⟩ '|' ⟨opt-ws⟩ ⟨ebnf-prod-list⟩

There is an ISO standard for EBNF. Our syntax and inclusion of features is not chosen to match the standard; it is what is convenient for our use.

4.11 Exercise – translating to EBNF

Translate this grammar from an earlier exercise to EBNF syntax.

A ⟶ B | C
B ⟶ aaB | ε
C ⟶ aaaC | ε

Then try to reduce the number of productions in the grammar, while maintaining the language defined.

Can you use only one production when using EBNF?

4.12 EBNF's syntactic sugar

EBNF gives us our first example of *syntactic sugar*; syntax that does not add new features to a language, only more convenient notation.

- As shown above, any grammar using the additional operators can be translated into one not using them.
  - But this likely requires more productions.
  - And certainly more characters/space on the page.

Syntactic sugar is a common feature of programming languages.

- Example: (imperative) languages often include various kinds of loops, where only one (or sometimes none!) is truly necessary.

When we discuss programming languages formally, we will usually omit constructs which are syntactic sugar.

- If anything, we may note how to represent them in a "core" language which includes less constructs.

4.13 Exercise – a very small language

Consider the following context-free language.

```
⟨stmt⟩   ::= ⟨assign⟩ | ⟨stmt⟩ "; " ⟨stmt⟩ | "while " ⟨expr⟩ " do " ⟨stmt⟩ | ⟨ws⟩ ⟨stmt⟩ ⟨w
⟨assign⟩ ::= ⟨var⟩ ⟨ws⟩ " := " ⟨expr⟩
⟨expr⟩   ::= ⟨var⟩ | ⟨const⟩ | ⟨expr⟩ ⟨op⟩ ⟨expr⟩ | ⟨ws⟩ ⟨expr⟩ ⟨ws⟩
⟨var⟩    ::= ('x' | 'y' | 'z') {⟨var⟩}
⟨const⟩  ::= (1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0) {⟨const⟩}
⟨op⟩     ::= '+' | '-' | '*' | '/' | '<' | '>' | '='
⟨ws⟩  ::= {' '} | {'\n'}
```

Provide some example programs in this language.
Consider trying to precisely describe the language in English.

4.14 Example – EBNF for C++

A good example of the practicality EBNF for specifying the syntax of languages is this EBNF grammar for C++ (presented in tabular form, rather than lists of productions as we use).

The grammar is much, much larger than anything we will write, but it is still quite concise for describing a real-world programming language.

5 Parsing and executable code

We will briefly summarise the parsing process.

- In this course, we are primarily interested in the beginning of this process, up to the construction of parse trees.
- The reality of

5.1 Lexemes and tokens

We have mentioned that both regular expressions and context-free grammars are used in the description of the syntax of programming languages.

However, up until this point, our example programming languages have been described exclusively by context-free grammars.

- Even the smallest syntactic units of the language have been described by the grammars.

  - For instance, we have used productions such as $\langle const \rangle ::= (1|2|3|4|5|6|7|8|9|0)\{\langle const \rangle\}$ which describes numerical constants.

This is not done in practice.
In practice,

- regular expressions are instead used to describe the smallest syntactic units of languages.
  - For example,
    - keywords such as `if` and `while`,
    - or constant values, such as `0` or ~"abc"~,
    - or names such as `height` or `sqrt`.
  - Lexemes cannot be broken down into meaningful pieces.
- Grammars are then used to describe the possible arrangements of lexemes.
  - The terminals of the grammar are then names for sets of lexemes, called *tokens*, rather than elements of Σ.
  - For instance,
    - the token `while` for the set containing only the keyword `while`,
    - or the token `int_literal` for the set $\{0, 1, -1, 2, ...\}$,
    - or the tohen `var` for the set of valid variable names.

5.2 The zeroth step – preprocessing

In some programming languages

5.3 The first step – lexical analysis

Parsing is the process of translating a program from plaintext to executable instructions

- whether this is done
  - ahead of time (compiling) or
  - when the program is to be run (interpreting),

  parsing is a necessary step before execution.

We now know the first step in parsing.

- Convert the plaintext source code into a sequence of tokens.
  - This process may be called *lexical analysis*, *lexing* or *tokenising*.
  - The program to carry this process out may be called a *lexer* or *tokeniser*.
  - Lexical analysis discards whitespace, comments, and any other irrelevant text.

5.4 The second step – parsing (syntactic analysis)

After converting from plaintext to a string of tokens, the next step of parsing is to construct the parse tree.

This step is part of the parsing process, but it is also usually called parsing.

- It may also be called *syntactic analysis*.

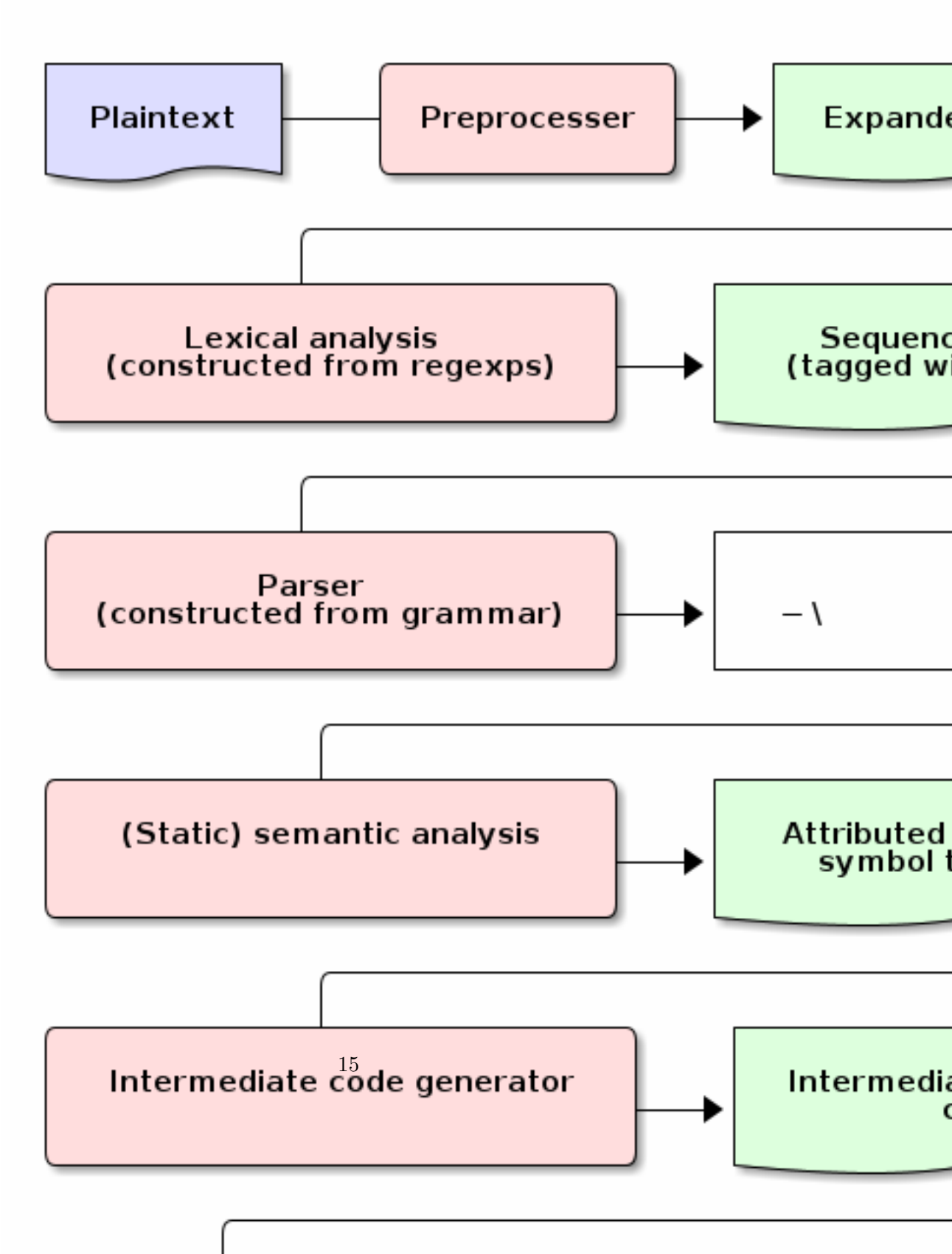
5.5 The third step – (static) semantic analysis

:TODO:

5.6 The fourth step – intermediate code generation

:TODO:

5.7 Visualising the entire parsing process



6 Ambiguity

6.1 Ambiguity

Recall that parsing a string (or deriving a string) using a grammar gives rise to a *parse tree* or *derivation tree*.

In many cases, there is more than one parse tree for a given string in the language produced by a grammar.

For instance, the string `aa` has four valid parse trees under the grammar

⟨A⟩ ::= a ⟨A⟩ | ⟨A⟩ a | ε

Exercise: find all four valid parse trees for `aa` with the above grammar.

6.2 Removing ambiguity

It is desirable to have a single parse tree for every program.

- We should not admit two syntactic interpretations for a program!

Three tools for removing ambiguity are

- requiring parentheses,
- introducing precedence rules, and
- introducing associativity rules.

6.3 Enforcing precedence and associativity with grammars

To enforce precedence using a grammar:

- Create a hierarchy of non-terminals.
- Higher-precedence operators are produced lower in the hierarchy.
- For instance,
  - An additive term can be a addition of multiplicative terms, which is an addition of literals, which can be the negation of a constant, variable or term.

To enforce associativity using a grammar:

- Left associative operators should be produced by left recursive non-terminals.
- And right associative operators by right recursive non-terminals.
- Operators of the same precedence must associate the same way!

6.4 Is addition associative?

Recall that addition is an associative operator.

- Meaning it is both left and right associative.

So the choice of whether addition in a language associates to the right or to the left may seem arbitrary.

- But numerical types in programming are not necessarily the same as numerical types in math!
- Addition of floating point numbers *is not associative*.
  - Consider a binary representation with two-digit coefficients.
  - $1.0_2 \times 2^0 + 1.0_2 \times 2^0 + 1.0_2 \times 2^2$ has a different value depending upon parenthesisation.

6.5 Abstract syntax

"Simple", ambiguous grammars do have a place in describing programming language syntax.

- Such grammars describe the *abstract syntax* of the language.
  - As opposed to *concrete syntax*.
- Consider programs as *trees* generated by the grammar for the abstract syntax of the language.
  - Trees do not admit ambiguity!
  - Such trees more efficiently represent programs.
    - The shape of the tree expresses structure.
    - Other unnecessary details may be left out.

6.6 Beyond context-free grammars: "static semantics"

For most interesting languages, context-free grammars are not quite sufficient to describe well-formed programs.

- They cannot express conditions such as "variables must be declared before use", and typing rules.
- It has been *proven* that CFGs are not sufficient.

  - At least some typing rules are possible to express, but prohibitively difficult.

Recall the Chomsky hierarchy of languages.

```
Regular  Context-free  Context-sensitive  Recursive
↪ Recursively enumberable
```

- The properties we need could be described by *context-sensitive* grammars.
  - But they are unwieldy!
- Instead, use *attribute grammars*; a relatively small augmentation to CFGs.
  - Each non-terminal and terminal may have a collection of *attributes* (named values).
  - Each production may have a collection of rules defining the values of the attributes and a collection of predicates reasoning about those attributes.

6.7 An example attribute grammar

Consider this simple grammar.

⟨S⟩ ::= ⟨A⟩ ⟨B⟩ ⟨C⟩
⟨A⟩ ::= ε | a ⟨A⟩
⟨B⟩ ::= ε | b ⟨B⟩
⟨C⟩ ::= ε | c ⟨C⟩

Suppose we want to allow only strings of the form $\texttt{a}^n\texttt{b}^n\texttt{c}^n$. There is no CFG that can produce exactly such strings. But we can enforce this condition using the above grammar augmented with attributes.

- Each of the non-terminals ⟨A⟩, ⟨B⟩ and ⟨C⟩ are given an attribute `length`.
- To each production with ⟨A⟩, ⟨B⟩ or ⟨C⟩ on the left side, we attach a rule to compute the `length`.
- The production ⟨S⟩ ::= ⟨A⟩ ⟨B⟩ ⟨C⟩ enforces the condition with a predicate.

```
⟨S⟩ ::= ⟨A⟩ ⟨B⟩ ⟨C⟩
Predicate: ⟨A⟩.length = ⟨B⟩.length = ⟨C⟩.length

⟨A⟩ ::= ε
Rule: ⟨A⟩.length ≔ 0

⟨A⟩1 ::= a ⟨A⟩2
Rule: ⟨A⟩1.length ≔ ⟨A⟩2.length + 1

⟨B⟩ ::= ε
Rule: ⟨B⟩.length ≔ 0

⟨B⟩1 ::= b ⟨B⟩2
Rule: ⟨B⟩1.length ≔ ⟨B⟩2.length + 1

⟨C⟩ ::= ε
Rule: ⟨C⟩.length ≔ 0

⟨C⟩1 ::= c ⟨C⟩2
Rule: ⟨C⟩1.length ≔ ⟨C⟩2.length + 1
```

In productions with multiple occurrences of the same non-terminal, we number the occurrences so we can easily refer to them in the rules/predicates.

7 Abstract and concrete syntax; ignoring ambiguity

8 The *semantics* of formal languages

The *semantics* of a language assigns a meaning to each sentence.

- In order to define a semantics, we must have in mind a *semantic domain*;
  - a domain of meanings into which we map sentences.
- For instance, if we are defining a language of natural numbers *Nat*, we will map sentences into the set $\mathbb{N}$.
- Or map elements of a languages of propositions into $\mathbb{B}$.

- We may often provide several different definitions of a particular mapping, to emphasise different details.

We may also have several semantic domains for a given language.

- In the case of programming languages, several domains of meaning have been proposed and used; the three most well known are
  - computing devices, whether a real-world machine or an *abstract* machine,
    - this is known as *operational semantics*
  - (mathematical) functions,
    - this is known as *denotational semantics*
  - precondition/postcondition pairs
    - this is known as *axiomatic semantics*

8.1 Example – semantics of a language of natural numbers

Consider again a language of terms intended to represent natural numbers.

⟨nat⟩ ::= "zero" | "suc" ⟨nat⟩

To assign meaning to these terms, we introduce a mapping from these (concrete) terms to (abstract) numerals.

```
eval zero = 0
eval (suc n) = (eval n) + 1
```

The evaluation function in this case is very obvious and trivial, because with this language is simply a concrete representation of the semantic domain.

- In comparison, when defining the semantics of programming languages, the language and the semantic domain are not so directly related.

8.2 Example – semantics of propositional logic

As a more complex example, we can map propositional logic terms into the set of booleans.

⟨prop⟩ ::= "tt" | "ff" | ¬ ⟨prop⟩ | ⟨prop⟩ (∧ | ∨ | ⇒ | ⟺)
↪ ⟨prop⟩

In order to make the mapping less trivial, let us define it without using boolean combinators; only constants and "if-then-else" statements.

```
eval tt = true
eval ff = false

eval (¬ p) = true    if eval p
             false   otherwise

eval (p ∧ q) = eval q   if eval p
               false    otherwise
```

...

Exercise: Complete this evaluation function.

8.3 Example – small-step semantics of propositional logic

The evaluation function defined above can be considered to be a *big-step* semantics.

- It is a (single-valued) relation between terms and their (final) value.

In contrast, we may define a *small-step* semantics

- which maps terms to terms which are "one step" simpler.
- Then, once we have reduced to a constant term, that may be mapped to a value (this part is not shown here).

```
reduce (¬ tt) = ff
reduce (¬ ff) = tt
reduce (¬ p)  = ¬ (reduce p)

reduce (tt ∧ q) = reduce q
reduce (ff ∧ q) = ff
reduce (p ∧ q)  = (reduce p) ∧ q
```

...

Exercise: Complete this reduction function.