Notes on Syntax and Parsing

When we describe a programming language, we distinguish between the *syntax* and the *semantics* of the language. The syntax describes the structure of a program (e.g., that code blocks can be nested and are enclosed by braces, that infix operators have a certain precedence and associativity, etc.). On the other hand, the semantics describes a program's meaning (i.e., what the program computes). In order to understand programming languages, it is important not to confuse these two concepts.

When we talk about the syntax of a programming language, we further distinguish between its *concrete syntax* and its *abstract syntax*. The concrete syntax defines which sequences of characters represent programs. The *abstract syntax* describes the structure of a program as an abstract syntax tree. Such a tree abstracts from some of the specifics of the concrete syntax, such as parenthesis in expressions, semicolons after statements, etc. The abstract syntax also makes the precedence and associativity of operators explicit.

The *parsing problem* is the problem of converting the program source code represented as sequences of characters (i.e., concrete syntax) into abstract syntax trees. You can learn more about this topic in a compiler course. Nevertheless, it is useful to have a basic understanding of how the concrete syntax of a programming language can be described and what the typical problems are when writing parsers for programming languages.

1 Formal Languages

In computer science, we formally describe languages as sets of words. Each word is a finite sequence of symbols drawn from a set that we call the *alphabet* of the language. For example, we can describe an arithmetic expression "3 + 5 * 8" as a word that is given by the sequence of symbols '3', '+', '5', '*', and '8'.

To describe languages in a compact form, we use *grammars*. A grammar is given by a set of rules, called *productions*. Productions tell us how the words of the language can be constructed from the symbols in the alphabet. For example, the following grammar describes the language of all arithmetic expressions:

$$
\begin{aligned}
&E \to E\,O\,E \\
&E \to (E) \\
&E \to x \qquad \text{where } x \in \mathbb{Z} \\
&O \to \texttt{+} \\
&O \to - \\
&O \to \texttt{*} \\
&O \to \texttt{/}
\end{aligned}
$$

Note that the third rule actually stands for an infinite set of productions (one

for each $x \in \mathbb{Z}$):

$$\begin{aligned} &\dots \\ E &\to -2 \\ E &\to -1 \\ E &\to 0 \\ E &\to 1 \\ E &\to 2 \\ &\dots \end{aligned}$$

We call the uppercase symbols that occur on the left-hand sides of productions, such as E and O, *nonterminal* symbols. The remaining symbols that are drawn from the alphabet of the language such as + and 3 are called *terminal* symbols.

Each grammar has an associated (nonterminal) starting symbol. In our example grammar, this is the symbol E. Starting from this symbol we apply the productions one by one until we obtain a word that consists only of terminal symbols. In each step, we pick one nonterminal in the current working word, choose a production in which this nonterminal occurs on the left-hand side, and replace the chosen nonterminal in the working word by the right-hand side of the chosen production. We call this process *derivation*.

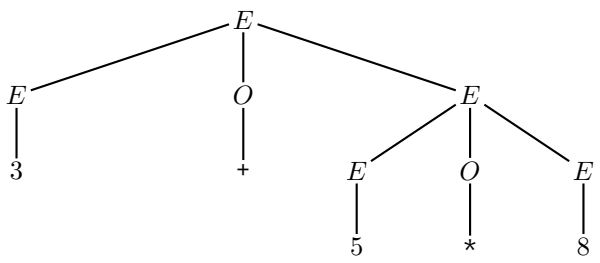
Using the above productions, we can derive words such as

$$\begin{aligned} &1 \\ &3 + 5 * 8 \\ &-14 / (42 + (0 - 1)) \end{aligned}$$

Here is a derivation of the word $3 + 5 * 8$:

$$\begin{aligned} E &\Rightarrow E\,O\,E \\ &\Rightarrow 3\,O\,E \\ &\Rightarrow 3 + E \\ &\Rightarrow 3 + E\,O\,E \\ &\Rightarrow 3 + 5\,O\,E \\ &\Rightarrow 3 + 5 * E \\ &\Rightarrow 3 + 5 * 8 \end{aligned}$$

Alternatively, we can represent this derivation by its *parse tree*:



The problem of constructing a parse tree for a given word in a language is the *parsing problem*. This problem can be solved automatically for the important class of *context-free languages*, which are the languages described by *context-free grammars*. In a context-free grammar, the left-hand sides of all productions consists of a single non-terminal symbol, as in our example above. Consequently, each derivation step rewrites a single nonterminal symbol in the working word regardless of the context in which this nonterminal occurs. The concrete syntax of most programming languages is context-free. So called parser generators can automatically construct parsers for context-free languages from a description of their grammar. We next define this class of languages and their associated grammars formally.

2 Context-Free Languages and Grammars

A *context-free grammar* is a tuple $G = (\Sigma, N, P, S)$ where

- Σ is a finite set of terminal symbols,
- N is a finite set of nonterminal symbols disjoint from Σ,
- $P \subseteq (N, (\Sigma \cup N)^*)$ is a finite set of productions, and
- $S \in N$ is the starting symbol.

We denote a production $(X, w) \in P$ by $X \to w$.

Let $G = (\Sigma, N, P, S)$ be a context-free grammar. For any two words, $u, v \in (\Sigma \cup N)^*$, we say v directly derives from u, written $u \Rightarrow v$, if there exists a production $X \to w$ in P and $u_1, u_2 \in (\Sigma \cup N)^*$ such that $u = u_1 X u_2$ and $v = u_1 w u_2$. That is, v is the result of applying $X \to w$ to u. We denote by $\Rightarrow^*$ the reflexive and transitive closure of the relation $\Rightarrow$ and we say that v derives from u if $u \Rightarrow^* v$.

The language of G, denoted $\mathcal{L}(G)$, is the set of all terminal words that can be derived from S:

$$\mathcal{L}(G) = \{\, w \in \Sigma^* \mid S \Rightarrow^* w \,\}$$

A language $\mathcal{L} \subseteq \Sigma^*$ is called context-free if it is the language of some context-free grammar G.

Note that the grammar for arithmetic expressions that we gave above is technically not a context-free grammar because the set of productions (as well as the set of terminal symbols) is infinite. For now, we will skim over this technicality. We will see later how we obtain a proper context-free grammar for arithmetic expressions.

3 Backus-Naur-Form

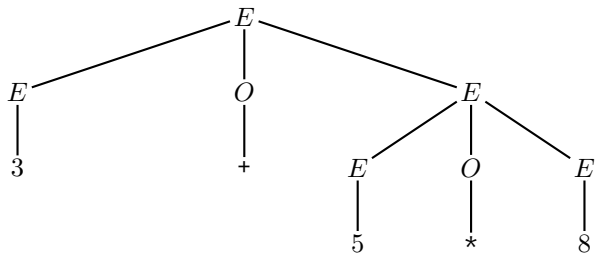
Often, context-free grammars are given in so-called *Backus-Naur-Form* (BNF). In this form, we use the symbol ::= instead of $\to$ to separate the two sides of

a production. Moreover, in a BNF, productions $X \to w_1, \ldots, X \to w_n$ for the same nonterminal symbol X can be summarized by a single rule $X ::= w_1 \mid \cdots \mid w_n$. For example, here is our grammar of arithmetic expressions in BNF:

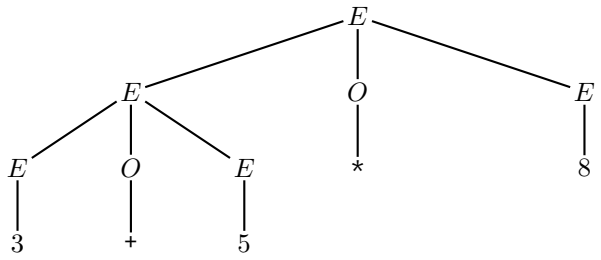
$$
\begin{aligned}
& x \in \mathbb{Z} \\
& \quad E ::= E\,O\,E \mid (E) \mid x \\
& \quad O ::= + \mid - \mid * \mid /
\end{aligned}
$$

4 Eliminating Ambiguity

Let us reconsider our grammar for arithmetic expressions and the derivation of the expression "3 + 5 * 8" given by the following parse tree:



This is not the only possible derivation of "3 + 5 * 8". Another one is given by the following parse tree:



We call a grammar in which a word has more than one derivation *ambiguous*. Ambiguity is a problem because the semantics of programs is given in terms of their abstract syntax trees, which are derived from parse trees. Typically, the semantics of a program depends on the structure of its parse tree. For example, with the canonical semantics of arithmetic expressions, the first parse tree of the expression "3 + 5 * 8" would evaluate to 43 whereas the second parse tree would evaluate to 64. Ideally, we would like to change our grammar so that the second parse tree no longer represents a valid derivation. This can be done by augmenting the grammar with additional disambiguation rules.

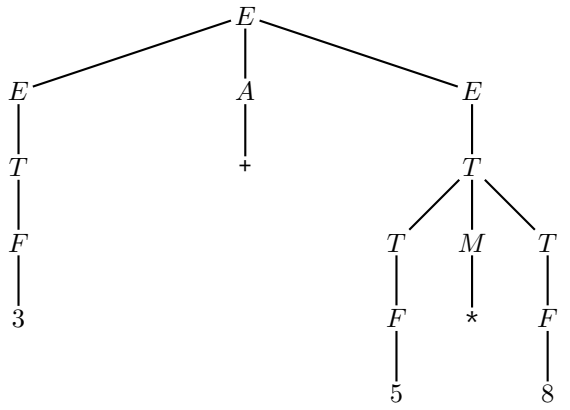
The problem of detecting whether a given context-free grammar is ambiguous is undecidable. Consequently, there does not exist a general algorithm that turns an ambiguous grammar into an unambiguous one. We therefore have to make

do with ad hoc techniques for resolving ambiguities. Fortunately, for grammars that describe programming languages, there exist some general recipes that work well in practice.

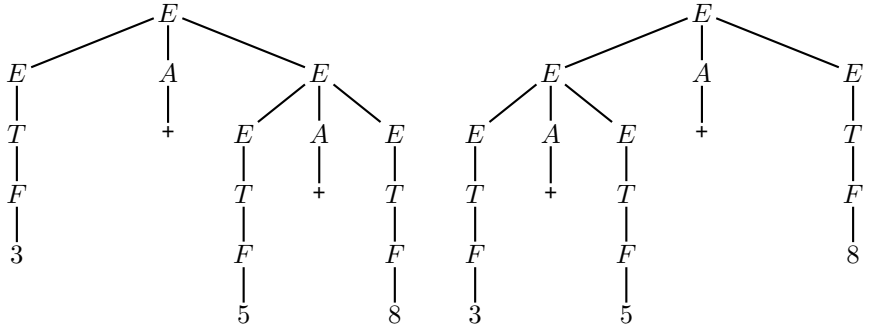
The ambiguity in our arithmetic expression grammar that we have observed for the expression "3 + 5 * 8" stems from the fact that the grammar does not distinguish between the additive operators, '+' and '-', and the multiplicative operators, '*' and '/'. We would like the multiplicative operators to *bind stronger* than the additive operators. We also say that the multiplicative operators have higher *precedence*. We can encode operator precedence by grouping expressions based on the types of operators and changing the productions so that expressions are expanded in the right order:

$$
\begin{aligned}
E &::= E\ A\ E \mid T \\
A &::= \texttt{+} \mid \texttt{-} \\
T &::= T\ M\ T \mid F \\
M &::= \texttt{*} \mid \texttt{/} \\
F &::= x \mid (E)
\end{aligned}
$$

Now the only valid parse tree for the expression "3 + 5 * 8" is the tree:



Our grammar is still ambiguous, though. For example, consider the expression "3 + 5 + 8". Here are two possible parse trees for this expression:



It seems that this ambiguity does not matter from a semantic point of view because addition on the integers is associative. That is, both trees would evaluate to 16. However, in computer programs we are normally working with bounded representations of integers. In this case, the arithmetic operations are often not associative due to potential arithmetic overflow. We would therefore like the operations to be parsed in a specific order. For example, arithmetic operators are usually defined as left-associative rather than right-associative, which means that the expression "3 + 5 + 8" should be parsed similar to "(3 + 5) + 8" rather than "3 +(5 + 8)".

To encode left-associativity of operators in our grammar, we can replace the right side of each binary expression by the base case of that expression type. This will force the repetitive matches of subexpressions onto the left side:

$$
\begin{aligned}
E &::= E\,A\,T \mid T \\
A &::= \texttt{+} \mid \texttt{-} \\
T &::= T\,M\,F \mid F \\
M &::= \texttt{*} \mid \texttt{/} \\
F &::= x \mid (E)
\end{aligned}
$$

5 Regular Languages

Finally, let us modify our grammar for arithmetic expressions so that it is actually context-free, i.e., so that the terminal symbols and productions are finite sets. We do this in two steps. First, we define a context-free grammar that describes integer numbers in decimal representation:

$$
\begin{aligned}
Z &::= \texttt{-}\,H \mid H \\
H &::= 0 \mid 1D \mid \cdots \mid 9D \\
D &::= \epsilon \mid 0D \mid \cdots \mid 9D
\end{aligned}
$$

The starting symbol of this grammar is Z and the terminal symbols are $\Sigma = \{\texttt{-}, 0, \ldots, 9\}$.

Next, we combine this grammar with the grammar:

$$
\begin{aligned}
E &::= E\,A\,T \mid T \\
A &::= \texttt{+} \mid \texttt{-} \\
T &::= T\,M\,F \mid F \\
M &::= \texttt{*} \mid \texttt{/} \\
F &::= Z \mid (E)
\end{aligned}
$$

to obtain a context-free grammar for arithmetic expressions.

The productions of our grammar for integer numbers have a special form. They all match one of the following shapes:

$$X ::= \epsilon$$
$$X ::= a$$
$$X ::= Y$$
$$X ::= aY$$

where X, Y are nonterminals and a is a terminal symbol. Grammars in which all productions are of these shapes form a special subclass of context-free grammars, called *regular grammars*. The languages of these grammars are correspondingly called *regular languages*. An equivalent and often more convenient way to describe such languages is by *regular expressions*.

Regular expressions R over an alphabet Σ themselves form a language, which can be described by a context-free grammar:

$R ::= \varepsilon$	empty string
$\mid a$	where $a \in \Sigma$
$\mid RR$	sequencing
$\mid R + R$	alternation
$\mid R^*$	repetition (Kleene star)
$\mid (R)$	grouping

Each regular expression R denotes a regular language $[\![R]\!]$ over Σ as follows:

- ε denotes the empty language $\emptyset$
- a symbol a from the alphabet denotes the language $\{a\}$ consisting of the singleton word a.
- a sequence of two regular expressions $R_1 R_2$ denotes $\{\alpha\beta \mid \alpha \in [\![R_1]\!], \beta \in [\![R_2]\!]\}$
- $R_1 + R_2$ denotes $[\![R_1]\!] \cup [\![R_2]\!]$
- repetition R^* denotes the set of words which are concatenations of zero or more strings from $[\![R]\!]$
- parentheses are used for grouping, i.e. (R) denotes $[\![R]\!]$

One often writes $R_1 \mid R_2$ instead of $R_1 + R_2$ for consistency with how alternation is expressed in BNF grammars. Also, the following shorthands are commonly used:

$$R^+ = RR^*$$
$$R^? = \varepsilon + R$$

Here is how we can write our language for integer numbers in decimal representation using a regular expression:

$$-^{?}(0 \mid (1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^{*}$$

Regular languages can be parsed more efficiently than general context-free languages. Compilers therefore split the parsing of the input program into two phases: a so-called *lexing phase* in which the character sequence representing the input program is converted into a token sequence, and the actual parsing phase in which the token sequence is converted into a parse tree. The tokens in the token sequence are subsequences of characters in the input program that have been grouped together, e.g., to form keywords of the language or numbers (as in the example of our arithmetic expression language). The program that takes care of the lexing phase is called *lexer* or *tokenizer*. The lexer is typically auto-generated from regular expressions, whereas the actual parser is auto-generated from a general context-free grammar.