Chapter 4

CONTEXT-FREE GRAMMARS AND PARSING

In the previous chapters, two equivalent ways of describing patterns were discussed. One was graph-theoretic, using the labels on paths in a kind of graph called an "automaton". The other was algebraic, using the regular expression notation. In this section, we look at a third, more powerful way of describing patterns using a recursive definition called a "context-free grammar".

Context-free grammars are important for the specification of programming languages. They provide a succinct notation for describing the syntax of typical programming language. Further, there are mechanical ways of turning a grammar into a "parser," one of the key components for a compiler for the language.

4.1 A Grammar for Arithmetic Expressions

Arithmetic expressions can be defined naturally with a recursive definition. We will consider arithmetic expressions that involve numbers, the four binary operators $+$, $-$, $*$ and $/$, and parentheses. The usual definition of such expression is an induction of the following form:

BASIS A number is an expression

INDUCTION If E_1 and E_2 are expressions, then each of the following is also an expression:

1. (E_1). That is, we may place parentheses around an expression to get a new expression
2. $E_1 + E_2$. That is, two expressions connected by a plus sign is also an expression. The following three rules cover connections using the other operators.

3. $E_1 - E_2$.
4. $E_1 * E_2$.
5. E_1/E_2.

Grammars allow us to write down such rules succinctly and with a precise meaning. Here is a grammar for the definition of arithmetic expressions above.

$$\langle \mathit{Expression} \rangle \rightarrow \mathit{number} \tag{4.1}$$
$$\langle \mathit{Expression} \rangle \rightarrow (\ \langle \mathit{Expression} \rangle\ ) \tag{4.2}$$
$$\langle \mathit{Expression} \rangle \rightarrow \langle \mathit{Expression} \rangle + \langle \mathit{Expression} \rangle \tag{4.3}$$
$$\langle \mathit{Expression} \rangle \rightarrow \langle \mathit{Expression} \rangle - \langle \mathit{Expression} \rangle \tag{4.4}$$
$$\langle \mathit{Expression} \rangle \rightarrow \langle \mathit{Expression} \rangle * \langle \mathit{Expression} \rangle \tag{4.5}$$
$$\langle \mathit{Expression} \rangle \rightarrow \langle \mathit{Expression} \rangle / \langle \mathit{Expression} \rangle \tag{4.6}$$

The symbol $\langle \mathit{Expression} \rangle$ is called a *syntactic category*; it stands for any string in the language of arithmetic expressions. The symbol $\rightarrow$ means “can be composed of”. For example, rule 4.2 states that an expression can be composed of a left parenthesis followed by any string that is an expression followed by a right parenthesis. Rule 4.1 is different because the symbol *number* on the right of the arrow is not intended to be a literal string, but a placeholder for any string that can be interpreted as a number.

There are three kinds of symbols that can appear in grammars. The first are “meta-symbols” that play a special role and do not stand for themselves. The $\rightarrow$ symbol is a meta-symbol, which is used to separate the syntactic category being defined from the strings from which it can be composed. The second kind of symbol is a syntactic category, which represents a set of strings being defined. The third kind of symbol is a *terminal*. Terminals can be characters, such as + or (, or they can be abstract symbols such as *number*, that stand for one or more strings that we may wish to define at a later time.

A *number* could be defined using a regular expression, as in

$$\begin{aligned} \mathit{digit} &= [\mathbf{0\text{-}9}] \\ \mathit{number} &= \mathit{digit}^+ \end{aligned}$$

The same idea can be expressed in grammatical notation, as in

$$\begin{aligned} \langle \mathit{Digit} \rangle &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ \langle \mathit{Number} \rangle &\rightarrow \langle \mathit{Digit} \rangle \\ \langle \mathit{Number} \rangle &\rightarrow \langle \mathit{Number} \rangle \langle \mathit{Digit} \rangle \end{aligned}$$

The above example also introduces the meta-symbol $|$, which abbreviates the ten productions

$$\begin{array}{rcl} \langle \mathit{Digit} \rangle & \rightarrow & 0 \\ \langle \mathit{Digit} \rangle & \rightarrow & 1 \\ & \ldots & \\ \langle \mathit{Digit} \rangle & \rightarrow & 9 \end{array}$$

We could similarly have combined the two productions for $\langle \mathit{Number} \rangle$ into one line.

This notation for describing grammars is sometimes referred to as *Backus-Naur Form* or BNF for short, after J. Backus and P. Naur who used similar notations to describe the grammars of Fortran and Algol 60 respectively.

4.2 A Grammar for Java Statements

We can describe the structure of Java control flow constructs using a grammar. The syntactic category $\langle \mathit{Statement} \rangle$ will be used to describe Java statements.

The first production describes a while-loop. That is, if we have a statement to serve as the body of the loop, we can precede it with the keyword while, an open parenthesis, a condition, and a close parenthesis.

$$\langle \mathit{Statement} \rangle \rightarrow \texttt{while(}\ \mathit{condition}\ \texttt{)}\ \langle \mathit{Statement} \rangle \tag{4.1}$$

Another way to build statements is using an if-construct. These constructs take two forms, depending on whether or not there is an else-part.

$$\langle \mathit{Statement} \rangle \rightarrow \texttt{if(}\ \mathit{condition}\ \texttt{)}\ \langle \mathit{Statement} \rangle \tag{4.2}$$

$$\langle \mathit{Statement} \rangle \rightarrow \texttt{if(}\ \mathit{condition}\ \texttt{)}\ \langle \mathit{Statement} \rangle\ \texttt{else}\ \langle \mathit{Statement} \rangle \tag{4.3}$$

Other constructs such as for-loops and switch statements are similar in spirit, and are left as exercises.

However, one other important formation rule is the block. A block uses the delimiters { and } around a list of zero or more statements. To describe blocks, we need an auxiliary syntactic category, which we call $\langle \mathit{StmtList} \rangle$. The productions for $\langle \mathit{StmtList} \rangle$ are

$$\langle \mathit{StmtList} \rangle \rightarrow \epsilon \tag{4.4}$$

$$\langle \mathit{StmtList} \rangle \rightarrow \langle \mathit{StmtList} \rangle\ \langle \mathit{Statement} \rangle \tag{4.5}$$

The first production is an an *empty* production; that is, a $\langle \mathit{StmtList} \rangle$ can be the empty string.

We can now define statements that are blocks as a statement list enclosed in curly braces:

$$\langle \mathit{Statement} \rangle \quad \rightarrow \quad \{ \, \langle \mathit{StmtList} \rangle \, \} \tag{4.6}$$

Finally, a Java statement can be an expression or a declaration followed by a semicolon. The grammar for Java expressions and declarations is left as an exercise.

$$\begin{aligned} \langle \mathit{Statement} \rangle \quad &\rightarrow \quad \langle \mathit{Declaration} \rangle \, ; \qquad (4.7) \\ \langle \mathit{Statement} \rangle \quad &\rightarrow \quad \langle \mathit{Expression} \rangle \, ; \qquad (4.8) \end{aligned}$$

4.2.1 Exercises

1. Give a grammar to specify the syntactic category $\langle \mathit{Identifier} \rangle$ of Java identifiers.

2. Add productions for $\langle \mathit{Statement} \rangle$ to include for-loops, do-loops, and switch-statements.

3. Extend the grammar of arithmetic expressions to include identifiers, method calls, and array indexes.

4. Give a grammar for Java expressions. Include assignment statements, method calls, *new*, *throw*, *case*-labels, *break*, *continue* and the conditional expression ? :.

5. Give a grammar for Java declarations.

4.3 Parse Trees

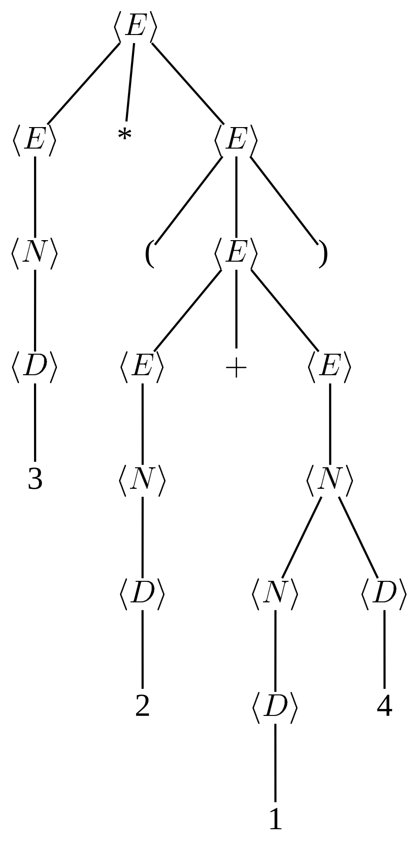
A given string belongs to the language generated by a grammar if it can be formed by repeated application of the productions. The grammar of arithmetic expressions below is used to illustrate this process.

$$\begin{aligned} \langle E \rangle \quad &\rightarrow \quad (\, \langle E \rangle \,) \mid \langle E \rangle + \langle E \rangle \mid \langle E \rangle - \langle E \rangle \mid \langle E \rangle * \langle E \rangle \mid \langle E \rangle / \langle E \rangle \mid \langle N \rangle \\ \langle N \rangle \quad &\rightarrow \quad \langle N \rangle \langle D \rangle \mid \langle D \rangle \\ \langle D \rangle \quad &\rightarrow \quad 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

The productions involved in generating the string `3*(2+14)` are as follows. One production is applied on each line, with the underlined syntactic category being expanded on the following line.

$$
\begin{array}{rcl}
\langle E \rangle & \rightarrow & \underline{\langle E \rangle} * \langle E \rangle \\
& \rightarrow & \underline{\langle N \rangle} * \langle E \rangle \\
& \rightarrow & \underline{\langle D \rangle} * \langle E \rangle \\
& \rightarrow & 3 * \underline{\langle E \rangle} \\
& \rightarrow & 3 * (\underline{\langle E \rangle}) \\
& \rightarrow & 3 * (\underline{\langle E \rangle} + \langle E \rangle) \\
& \rightarrow & 3 * (\underline{\langle N \rangle} + \langle E \rangle) \\
& \rightarrow & 3 * (\underline{\langle D \rangle} + \langle E \rangle) \\
& \rightarrow & 3 * (2 + \underline{\langle E \rangle}) \\
& \rightarrow & 3 * (2 + \underline{\langle N \rangle}) \\
& \rightarrow & 3 * (2 + \underline{\langle N \rangle}\langle D \rangle) \\
& \rightarrow & 3 * (2 + \underline{\langle D \rangle}\langle D \rangle) \\
& \rightarrow & 3 * (2 + 1\underline{\langle D \rangle}) \\
& \rightarrow & 3 * (2 + 14)
\end{array}
$$

A *parse tree* is a concise representation of these productions:



Every interior node v in a parse tree represents the application of a production. I.e., there must be some production such that

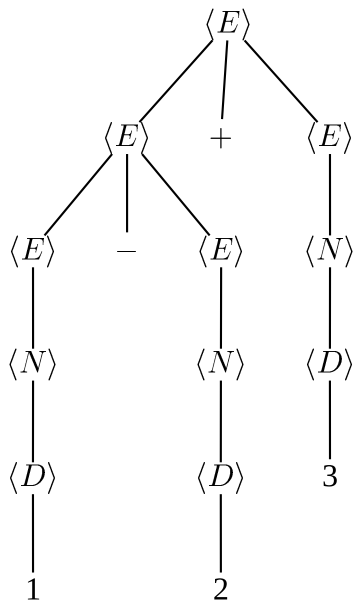
1. The syntactic category labelling v is the head of the production, and
2. The labels of the children of v, from the left, form the body of the production.

In the example above, the root and its children represent the production

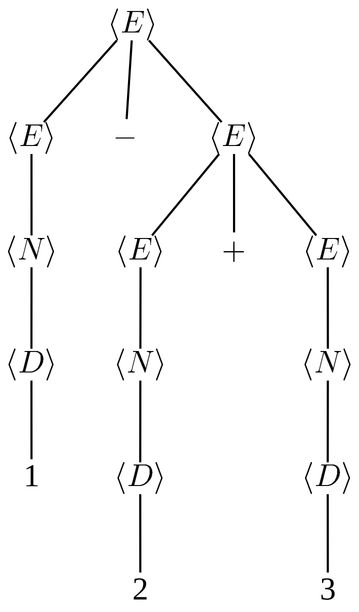
$$\langle E\rangle \quad \rightarrow \quad \langle E\rangle * \langle E\rangle$$

4.4 Ambiguity and the Design of Grammars

Consider the expression $1-2+3$. It has two parse trees, depending on whether we group operators from the left or from the right.



(a) Correct parse tree



(b) Incorrect parse tree

This ambiguity is related to the *associativity* of operators. Conventionally, a sequence of expressions combined with the operators `+` and `-` is evaluated left-to-right, so that `1-2+3` is equivalent to `(1-2)+3`.

Another form of ambiguity arises with operators of different *precedence*. Convention has it that multiplication and divisions are done before additions and subtractions, so that `1+2*3` is equivalent to `1+(2*3)`.

Sometimes ambiguity makes no difference. All the parse trees for the expression `1+2+3+4` are equivalent for the purposes of evaluating the sum. However, this is not true in general.

Aside: Of course, fully parenthesizing expressions obviates the need for conventions regarding associativity and precedence, as does adopting an unambiguous notation such as reverse polish. The language Lisp adopts the former approach, and the Forth group of languages (a group that includes Postscript) adopt the latter.

4.5 An Unambiguous Grammar for Expressions

It is possible to construct an unambiguous grammar for arithmetic expressions. The "trick" is to define three syntactic categories, with the following intuitive meanings:

1. $\langle Factor \rangle$ generates expressions that cannot be "pulled apart," that is, a factor is either a single operand or any parenthesized expression.

2. $\langle Term \rangle$ generates a product or quotient of factors. A single factor is a term, and thus is a sequence of factors separated by the operators `*` or `/`. For example, `12*2/5` is a term.

3. $\langle Expression \rangle$ generates a sum or difference of one or more terms. A single term is an expression, and thus is a sequence of terms separated by the operators `+` or `-`. Examples of expressions are `12`, `12/3*45` and `12+3*45-6`.

The grammar is given below, using the shorthands $\langle E \rangle$ for $\langle Expression \rangle$, etc.

$$\langle E \rangle \rightarrow \langle E \rangle + \langle T \rangle \mid \langle E \rangle - \langle T \rangle \mid \langle T \rangle \tag{4.1}$$
$$\langle T \rangle \rightarrow \langle T \rangle * \langle F \rangle \mid \langle T \rangle / \langle F \rangle \mid \langle F \rangle \tag{4.2}$$
$$\langle F \rangle \rightarrow (\ \langle E \rangle\ ) \mid \langle N \rangle \tag{4.3}$$
$$\langle N \rangle \rightarrow \langle N \rangle \langle D \rangle \mid \langle D \rangle \tag{4.4}$$
$$\langle D \rangle \rightarrow \mathsf{0} \mid \mathsf{1} \mid \mathsf{2} \mid \mathsf{3} \mid \mathsf{4} \mid \mathsf{5} \mid \mathsf{6} \mid \mathsf{7} \mid \mathsf{8} \mid \mathsf{9} \tag{4.5}$$

To see how the grammar works, consider how the expression `1-2+3` can be parsed. The original, ambiguous grammar had the option of choosing for the first production either $\langle E \rangle \rightarrow \langle E \rangle - \langle E \rangle$ or $\langle E \rangle \rightarrow \langle E \rangle + \langle E \rangle$. The former parses `1` and `2+3` as expressions, while the latter parses `1-2` and `3` as expressions.

Our new grammar makes it clear that the $\langle E \rangle \rightarrow \langle E \rangle - \langle T \rangle$ production cannot be used, since `2+3` cannot be parsed as a term. The only option left is to use the production $\langle E \rangle \rightarrow \langle E \rangle + \langle T \rangle$, taking `3` as the $\langle T \rangle$.

Note how the causes of ambiguity, associativity and precedence, are resolved.

associativity A production of the form $\langle E \rangle \rightarrow \langle E \rangle \oplus \langle T \rangle$ generates a left-associative expression. Changing this to $\langle E \rangle \rightarrow \langle T \rangle \oplus \langle E \rangle$ makes the operator right-associative, so that $1 \oplus 2 \oplus 3$ is parsed as $1 \oplus (2 \oplus 3)$.

To make a non-associative expression (i.e., one in which $a \oplus b \oplus c$ is not valid), use a production of the form $\langle E \rangle \rightarrow \langle T \rangle \oplus \langle T \rangle$

precedence The distinction among expressions, terms and factors enforces the correct grouping of operators at different levels of precedence. For example, the expression `1-2*3` has only one parse tree, which groups the subexpression `2*3` first.

4.5.1 Exercises

1. Using the unambiguous grammar for arithmetic expressions, give the unique parse tree for the following expressions:

   (a) $(1 + 2)/3$

   (b) $1 * 2 - 3$

   (c) $(1 + 2) * (3 + 4)$

2. Extend the unambiguous grammar to include an exponentiation operator, `^`, which is at a higher level of precedence than `*` and `/`. Do this by introducing a new syntactic category *primary* to be an operand or parenthesized expression, and re-define a *factor* to be one or more primaries connected by the exponentiation operator. Note that exponentiation groups from the right, not the left, so that `2^3^4` means `2^(3^4)`. How do we force grouping from the right among primaries?

3. Extend the unambiguous grammar to allow the comparison operators `=`, `<=`, etc., which are all at the same level of precedence. That precedence is below that of `+` and `-`, so that `1+2<3` is equivalent to `(1+2)<3`. Note that the comparison operators are *non-associative*; that is, `1<2<3` is not a legal expression.

4. Extend the unambiguous grammar to include the unary minus sign. This operator is at a higher precedence to any other operator, so that `-2*-3` is grouped as `(-2)*(-3)`.

5. Extend the unambiguous grammar to include the logical operators &&, `||` and `!`. Give `||` a lower precedence to &&, and make them both lower precedence to the comparison operators `=`, `<=`, etc.. Thus, `1<2&&3>4||1=4` parses as

   `((1<2)&&(3>4))||(1=4)`

4.6 Constructing Parse Trees

Grammars are similar to regular expressions in that both notations describe languages but do not give directly an algorithm for determining whether a string is in the language being defined. For regular expressions, we have seen how to convert a regular expression into a nondeterministic finite automaton and then to a deterministic one; the latter can be implemented directly, as a program.

There is a somewhat analogous process for grammars. However, grammars are a more expressive notation than regular expressions, and we cannot, in general, convert a grammar into a deterministic finite automaton. However, it is often possible to convert a grammar to a program that, like an automaton, reads the input from beginning to end and judges whether the input string is in the language of the grammar. The most important such technique, called “LR parsing,” is beyond the scope of this paper.

Instead, we will look at a simpler, but less powerful parsing technique called “recursive descent,” which uses a collection of mutually recursive functions, each corresponding to one of the syntactic categories of the grammar.

We will build a recursive descent parser for a simple grammar of balanced parentheses, given below.

$$\langle B \rangle \;\rightarrow \tag{4.1}$$

$$\langle B \rangle \;\rightarrow\; (\langle B \rangle)\,\langle B \rangle \tag{4.2}$$

Production 4.1 states that an empty string is balanced.

Production 4.2 states that one way to find a string of balanced parentheses is to fulfill the following four goals in order:

1. Find the character `(`, then
2. Find a string of balanced parentheses, then
3. Find the character `)`, and finally
4. Find another string of balanced parentheses.

An *input cursor* keeps track of the next character in the input stream. We define a method `peek` to return this character. The end of the input is marked by a special *endmarker*, which indicates that the entire string has been read. A method `next` advances to the next input character. If the input is a string, `s`, we can use an integer input cursor, `pos`, and define these methods as follows:

```
void next( ) { pos++; }

char peek( ) {
  return pos < s.length( ) ? s.charAt( pos ) : (char)0;
}
```

It is useful to define a method `check` to test whether the next character matches a given terminal.

```
void check( char c ) throws ParseError {
  if( peek( ) != c )
```

```
    throw new ParseError( s, pos, "Expected '" + c + "'" );
  next( );
}
```

The result returned by the parser is a tree, which requires a constructor for each production in the grammar. The complete source code can be found on the COMPSCI 220FT WWW page.

```
Tree parseB( ) throws ParseError {
  if( peek( ) == '(' ) {
    next( );
    Tree b1 = parseB( );
    check( ')' );
    Tree b2 = parseB( );
    return new Tree( "B -> ( B ) B", b1, b2 );
  } else
    return new Tree( "B -> " );
}
```

4.6.1 Limitations of recursive descent

Recursive descent can be applied to many grammars, but not to all. This can be observed with the unambiguous grammar for arithmetic expressions. Naively coding the productions for this grammar as mutually recursive functions will result in an infinite recursion on each of the left-recursive rules. Furthermore, even if a grammar does not have left-recursive rules, the recursive descent method may be unsuitable. The basic requirement is that, for each syntactic category $\langle S\rangle$ that has more than one production, we need to be able to select the correct production for $\langle S\rangle$ by looking at only the next terminal (the *lookahead* symbol).

It is possible to contort the unambiguous grammar for arithmetic expressions into a form that allows it to be parsed using recursive descent, but the process requires introducing a large number of non-intuitive syntactic categories. Faced with such a grammar, the correct course of action is invariably to use a more powerful "bottom up" parsing method, such as LR-parsing. However, such methods are beyond the scope of this paper.

4.6.2 Exercises

1. Show the sequence of method calls made by `parseB` on the inputs
   (a) `(())`
   (b) `(()())`

(c) `())(`

2. The following grammar defines non-empty lists, which are elements separated by commas and surrounded by parentheses. An element can be either an atom or a list structure. Here, $\langle E\rangle$ stands for element, $\langle L\rangle$ for list, and $\langle T\rangle$ for "tail," that is, either a closing `)`, or pairs of commas and elements ended by `)`.

   Write a recursive descent parser for this grammar.

$$\langle L\rangle \rightarrow (\ \langle E\rangle\ \langle T\rangle \tag{4.1}$$
$$\langle T\rangle \rightarrow ,\ \langle E\rangle\ \langle T\rangle \tag{4.2}$$
$$\langle T\rangle \rightarrow\ ) \tag{4.3}$$
$$\langle E\rangle \rightarrow \langle L\rangle \tag{4.4}$$
$$\langle E\rangle \rightarrow atom \tag{4.5}$$

3. (*) Write a recursive descent parser for the grammar of Java statements given in section 4.2. Start by writing a *tokeniser* that recognises Java reserved words, identifiers, number, string and character constants, and symbols (parentheses, braces, etc.). Define a suitable `Tree` class to return the parse tree.

4.7 Tokenizing the Input Stream

Keen observers may have noticed that none of the grammars given in these notes are able to cope with the presence of white space or comments in the input. For example, the parser for balanced parentheses will *fail* on the input string `"(␣)"` (try it!). Re-writing the grammar to permit white space and comments results in not only a more complicated grammar but parse trees that contain a potentially large number of uninteresting nodes.

To avoid these problems, it is usual to *tokenize* the input to the parser, using a finite automaton. A finite automaton can be used to identify substrings in the input that are of interest to the parser, and return a code (usually a small integer) for the token type. Token types typically include "integer constant", "identifier", "open parenthesis", etc. The actual token can be placed by the automaton in temporary storage for the parser to retrieve when building the parse tree.

For example, given the input `for( i = 0; i < 100; i++ )`, the finite automaton might return a sequence of codes "reserved word *for*", "open parenthesis", "identifier", "equals", "integer constant", "semicolon", etc.

Finite automaton are powerful enough to be able to distinguish between white space and comments that appears in strings (and must therefore be preserved) and white space and comments appearing elsewhere (which it can ignore).