4 Syntax analysis

There exists a large number of techniques for parsing context free grammars. Most programming languages are deliberately designed to be describable by such grammars to take advantage of these techniques. This raises two questions.

- What is a context free grammar?
- Why do we want to parse programs, i.e. identify the syntactic components of them?

We concentrate on considering the answer to the former question in some detail, but a short reply to the latter will help to motivate our studies.

We parse programs because the semantics (sequences of actions or definitions) of programming languages are attached to syntactic structures. In other words, a certain grammatically defined sequence of strings may map on to a certain activity or sequence of activities. The semantics of a language can be viewed as attributes of its syntactic structures.

In general we wish first to parse all or part of a program, then perform certain transformations, such as converting infix to postfix expressions, and finally derive the corresponding actions, as an intermediate or machine code representation.

With our motives clearer, we begin with the definition and analysis of context free grammars. We assume lexical analysis delivers up abstract tokens for our use.

Context free grammars

Chomsky recognised a hierarchy of grammars and numbered then 0, 1, 2 and 3. This moves from grammars defined very freely (type 0) to those defined in very restrictive ways. A means of describing the differences is in terms of production rules.

A type 0 grammar imposes no restrictions on left hand sides of productions, except that no terminal symbols should appear there. In particular more than one non-terminal symbol may be given.

A type 1, context dependent, grammar imposes the restriction that each production replaces only one of its left hand side non-terminals in its right hand side. Thus that production for that non-terminal depends on its being in the context provided by the other left hand side non-terminal symbols.

A type 2, context free, grammar allows only one non-terminal symbol on the left hand side of each production. This gives the familiar Backus Normal or Backus Naur Form notation. Context free grammars allow much simpler parsing, as the identification of a single non-terminal allows us to recognise a syntactic construction.

A type 3, regular definition, grammar has been defined under lexical analysis.

Note that several grammars may define the same language, each supporting a particular parsing scheme.

Notations and BNF

There are several formats or notational conventions associated with BNF. They are all equivalent and are a product of improvements in the character sets and text processing facilities available to computer users, combined with a search for concise expression of ideas. They all seek to present four fundamental components in terms of a meta language. These are

- The production operator, usually one of ::=, ⇒ and →. I shall use →
- Non-terminal symbols, usually `<name>` or (italicised) *name*. I shall use *name*
- Terminal symbols, i.e. those identified by the lexical analyser usually one of `<id>` or (boldface, roman) **id**. I shall use **id**.
- Literal strings, i.e. characters and sequences of characters, usually **begin** or "begin". I shall use **begin**.

In fact the lexical analyser will normally have substituted a lexical value for both literal strings and terminal symbols and the sequence **begin** is used within the syntax analyser as a mnemonic for the corresponding lexical token.

Productions define the set of strings which can replace a non-terminal symbol. In a complete language description there must be a sequence of productions (⇒) leading to literal strings for every non-terminal. As we have seen, this may be unnecessary for the syntax analysis phase, since some productions are regular expressions suitable for lexical analysis. Thus all terminals in the syntactic phase are really lexical tokens.

A concatenation operation is implied by the order of symbols on the right hand side of a production. Note that this allows us to treat the terminal **begin** as either a single value, representing an identifier/keyword, or as five values, representing the characters, **b**, **e**, **g**, **i** and **n**. The interpretation depends on our lexical analysis strategy.

The repetition operators `*`, `+` and `?` may be used in the same way as for regular expressions. Operators and parentheses in the meta language will be written in teletype font, e.g. `(` and `)`, where this helps to avoid confusion with the corresponding terminal symbols, **(** and **)**.

The alternative operator is also the same as for regular expressions, i.e. `|`, and the same precedence is assumed.

Any language definition or *grammar* which is useful has a single start non-terminal, from which any string in the language may be derived as part of a legal set generated by a sequence of productions.

Views of productions

There are two ways of interpreting a production rule. One is to use it to derive a string, which is an instance of the left hand side's non-terminal symbol, by successively replacing the non-terminals on the right hand side according to the production rules of those non-terminals. This expansion is equivalent to *top down* parsing.

The other view is to use the production rules to 'collapse' an existing string back to a single non-terminal, which, for a complete set of strings, e.g. a complete program, would leave just the start symbol of the language. This is equivalent to *bottom up* parsing.

Types of grammar

There are a number of types of grammar, defined by their properties. It is quite common to be able to describe one language by several grammars with different properties. In order to use a particular parsing method it is often necessary to convert an existing grammar to one with the required properties.

Left and right recursion

One basic division of productions is into those which define sequences of symbols with and those which define them without without left or right recursion. Note that it is quite possible to have a mixture of both kinds of production within the same grammar.

An example of an *immediately* left recursive definition is

$$\textit{expression} \rightarrow \textit{expression}\ (\ +\ |\ -)\ \textit{term}\ |\ \textit{term}$$

The general form is

$$A \rightarrow A\ S\ |\ B$$

Note that there must be a non-recursive alternative, otherwise the recursion is unbounded.

Because of the danger of non-terminating recursion, left recursive grammars can never be used in top down parsers.

On the other hand the following production is non-recursive.

$$\textit{expression} \rightarrow \textit{term}\ \textit{expression\_end}$$

and can be used, with the following production, to generate the same strings.

$$\textit{expression\_end} \rightarrow (+\ |\ -)\ \textit{term}\ \textit{expression\_end}\ |\ \epsilon$$

This production is right recursive and is safe for a top down parser. This transformation can be generalised as

$$A \rightarrow A\ S\ |\ B$$

is equivalent to

$$A \rightarrow B\ C$$

$$C \rightarrow S\ C\ |\ \epsilon$$

Left recursion does not have to be immediate, of course. A chain of productions may lead to a recursive definition. In these cases a more complex problem is presented. However, as long as there are no cycles or ϵ productions in a grammar algorithms exist to eliminate indirect left recursion also. In practice both cycles and ϵ productions can be removed prior to left recursion elimination. See Aho, Sethi and Ullman p177 for details.

Eliminating ambiguity

Recursion in a grammar can introduce a sort of ambiguity into the parsing of a string. More obvious ambiguities can arise without recursion being involved at all, however. Consider the classic problem of the "dangling else".

stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	*other*

This grammar produces a single parse tree for the string:

```
if E1 then S1 else if E2 then S2 else S3
```

but two possible parsings of:

```
if E1 then if E2 then S1 else S2
```

Pascal forbids the use of a conditional sentence immediately following the keyword `then`, to avoid this problem. However, it is possible to produce an unambiguous grammar for a language, such as Algol 60, which permits it. In Algol the `else` is assumed to match the closest preceding `then`.

For this interpretation, the following grammar works:

stmt	→	*matched_stmt*
	|	*unmatched_stmt*
matched_stmt	→	**if** *expr* **then** *matched_stmt* **else** *matched_stmt*
	|	*other*
unmatched_stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *matched_stmt* **else** *unmatched_stmt*

The same strings are possible, but with only one parsing.

Associativity of operators

Some operators in a language associate to the left, others to the right. This is a property of the language not the grammar. It is invariant across all grammars describing the same language. It is, however, easier to deal with left association in languages by using grammars which are not right recursive.

An operator is said to associate to the left it, when, if it occurs on both sides of an operand within an expression, that operand is "taken by" the operator to its left. Thus, because - (minus) is normally left associative,

$$\begin{aligned} 7-2-3 &= (7-2)-3 \\ 7-2-3 &\neq 7-(2-3) \end{aligned}$$

Exponentiation, **, is often right associative, i.e.

$$2 * *3 * *4 \quad = \quad 2 * *(3 * *4)$$

We shall see this in our discussion of parsing techniques.

Context or prediction

Although productions define a context free grammar when only a single, non-terminal symbol exists on each left hand side, knowledge of the sequence which occured to the current sub-string's immediate left can be very helpful. Grammars are needed which can be scanned left to right and where only the left context is needed to simplify parsing, i.e. reduce the amount of backtracking. The names given to such grammars reflect this by beginning with L.

The most extreme form is the predictively parsable grammar, where each left hand side has only one possible right hand side in every production. Grammars of this sort may be parsed top down with no backtracking.

Derivation

The choice of derivation in parsing is leftmost or rightmost. This defines which non-terminal on the right hand side of a production will be replaced first when deriving a string from it. If the leftmost is to be replaced first, the grammar is said to have a leftmost derivation and vice versa.

The second letter of the general classes of grammar most widely used in parsing expresses the choice of derivation. Thus, LL is left context, leftmost derivation and LR is left context, rightmost derivation.

LL grammars are suitable for top down parsing, LR grammars are typically parsed bottom up.

Lookahead

The final general division is in terms of the number of symbols that a parser would need to look ahead to parse a grammar. This must be 1 for a predictive parser and so a grammar suitable for top down, predictive analysis, i.e. one with a distinct start symbol in each alternative on the right hand side of each production, is called an LL(1) grammar.

SLR and LALR

LR grammars are further divided into those parsable from tables generated by simple techniques, SLR grammars, and those parsable by more complex tables and requiring varying degrees of lookahead, LALR grammars.

Operator grammars

An operator grammar has no two adjacent non-terminals in any of its right hand sides, i.e. there is always a non-empty sequence of terminals separating the non-terminals. The grammar then defines precedences for all terminals and allows a parsing method known as operator precedence parsing.

Very few languages can be described in this way, but one important part of most languages, expressions, can. Thus, it is common to define expressions as an operator sub-language within

an LL(1) language. Many recursive descent compilers use operator precedence parsing for their expression analysis.

An example of an operator grammar is

$$E \rightarrow E\ A\ E \mid \texttt{(}\ E\ \texttt{)} \mid \texttt{-}E \mid \texttt{id}$$

$$A \rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\^{}}$$

Precedence can then be assigned using the disjoint relational symbols.

$$\begin{aligned} \gtrdot &- \text{ has precedence over} \\ \lessdot &- \text{ yields precedence to} \\ \doteq &- \text{ has equal precedence with} \end{aligned}$$

This need not be done for every pair of terminals.