Syntax

1 Grammars

Once upon a time the textbook for MACM 101 was *Discrete Mathematics and Its Applications*, 4th edition, by K.H. Rosen, McGraw Hill, 1999, and the course content included regular and context-free grammars.

I will use some of the definitions and notations from Rosen in these lectures.

Notational Note: Notation, and to some extent definitions, differ from one book to another.

On page 631 of Rosen we find:

DEFINITION 1. A *vocabulary* (or *alphabet*) V is a finite, nonempty set of elements called *symbols*. A *word* (or *sentence*) over V is a string of finite length of elements of V. The *empty string* or *null string*, denoted by λ, is the string containing no symbols. The set of all words over V is denoted by V^*. A *language over* V is a subset of V^*.

For example, V might be the set of ASCII (or Unicode) characters, so V^* would be the set of possible contents of a text file using this character set.

In *Haskell* terms, if V is the set of values for type `Char`, perhaps excluding $\perp_{Char}$, then V^* would be the set of *Haskell* strings, excluding partial and infinite strings, and perhaps excluding strings containing $\perp_{Char}$.

If $u, v \in V^*$ for some alphabet V, then we write uv for the *concatenation* of u and v. Concatenation corresponds to appending in *Haskell*, so in *Haskell* we would write `u++v` instead of uv.

We will generalize the concept of concatenation to sets of strings (languages). If $A, B \subseteq V^*$ then $AB = \{ab \mid a \in A, b \in B\}$.

When appropriate, we will interpret (coerce) a symbol to be a string of length one, or interpret a string to be a singleton set of strings (a one word language). Hence, a symbol may denote the symbol itself, a word or a language, depending on context.

Let $A \subseteq V^*$ be any set of strings. We define

$$\begin{aligned} A^0 &\equiv \{\lambda\} \\ A^1 &\equiv A \\ &\ldots \\ A^{k+1} &\equiv AA^k \end{aligned}$$

$$\begin{aligned} A^+ &\equiv \bigcup_{k=1}^{\infty} A^k \\ A^* &\equiv \bigcup_{k=0}^{\infty} A^k \end{aligned}$$

Notice that $A^* \equiv \{\lambda\} \cup A^+$ and that V^*, as previously defined, is just A^* with $A = V$.

On this page of these notes we find:

DEFINITION: A *production* is a relation on V^*. We write $u \rightarrow v$ to mean $(u, v) \in P \subseteq V^* \times V^*$.

Returning to page 631 of Rosen we find:

DEFINITION 2. A *phrase-structured grammar* $G = (V, T, S, P)$ consists of a vocabulary V, a subset T of V consisting of the terminal elements, a start symbol S from V, and a set of productions P. The set $V - T$ is denoted by N. Elements of N are called *nonterminal symbols*. Every production in P must contain at least one nonterminal on the left side.

In practice, we will be interested only in productions in $N \times V^*$. That is, in the grammars that we will study, productions will always have only a single nonterminal (and no terminals) on the left side.

From page 632 of Rosen we have:

DEFINITION 3. Let $G = (V, T, S, P)$ be a phrase-structured grammar. Let $w_0 = lz_0r$ (that is, the concatenation of l, z_0, and r) and $w_1 = lz_1r$ be strings over V. If $z_0 \rightarrow z_1$ is a production of G, we say that w_1 is *directly derivable* from w_0 and we write $w_0 \Rightarrow w_1$. If $w_0, w_1, \ldots, w_n, n \geq 0$, are strings over V such that $w_0 \Rightarrow w_1, w_1 \Rightarrow w_2, \ldots, w_{n-1} \Rightarrow w_n$, then we say that w_n is *derivable from* w_0, and we write $w_0 \overset{*}{\Rightarrow} w_n$. The sequence of steps used to obtain w_n from w_0 is called a *derivation*.

Still on page 632, (with a couple of minor errors fixed), Rosen says:

DEFINITION 4. Let $G = (V, T, S, P)$ be a phrase-structured grammar. The *language generated by* G (or the *language of* G), denoted by $L(G)$, is the set of all strings of terminals that are derivable from the start symbol S. In other words,

$$L(G) = \{w \in T^* | S \overset{*}{\Rightarrow} w\}.$$

1.1 Regular Languages

A *regular grammar* (or Chomsky type 3 grammar) is one where every production is either the production $S \rightarrow \lambda$, or the left side is an element of N and the right side is an element of $N \cup (TN) \cup T$. (That is, the left side is a single nonterminal and the right side is either a nonterminal, a terminal, or a terminal followed by a nonterminal. Note, in particular, that a nonterminal followed by a terminal is not allowed.)

The language generated by a regular grammar is called a *regular language* .

Notational Note: Some books do not allow a regular language (or a context-free languages, considered later) to include the empty string, λ.

1.1.1 Regular Expressions

A regular language can also be defined by a *regular expression* .

Many text editors and several programming languages support regular expressions. Unfortunately, these almost always use slightly different definitions of regular expressions, mainly because of extra features. Furthermore, these extra features usually mean that some languages that are not regular can be defined, in addition to all actual regular languages.

That is, regular expressions in software tools are not regular expression at all. It is rather like British public schools being private schools, or Algol not being an Algol-like language. A definition has evolved away from its original meaning.

DEFINITION: A *regular expression* over an alphabet V is

1. λ,
2. $\emptyset$,
3. a for any terminal symbol $a \in T$,
4. $(R_1 R_2)$ where R_1 and R_2 are any two regular expressions,
5. $(R_1 | R_2)$ where R_1 and R_2 are any two regular expressions, and
6. $(R)^*$ where R is any regular expression.

To reduce the number of parentheses we will assume that * binds more tightly than concatenation and catenation binds more tightly than $|$.

For example,

$$ab|cd^* \quad \text{means} \quad ((ab)|(c(d^*)))$$

A regular expression denotes a language in the obvious way.

We can formally define the meaning of a regular expression as follows.

$$
\begin{aligned}
&M[\![\lambda]\!] = \{\lambda\}\\
&M[\![\emptyset]\!] = \emptyset\\
&M[\![a]\!] = \{a\}\\
&M[\![(R_1R_2)]\!] = (M[\![R_1]\!])(M[\![R_2]\!])\\
&M[\![(R_1|R_2)]\!] = M[\![R_1]\!] \cup M[\![R_2]\!]\\
&M[\![(R^*)]\!] = (M[\![R]\!])^*
\end{aligned}
$$

where R, R_1 and R_2 are arbitrary regular expressions and a in any element of V.

It turns out the a language can be defined by a regular expression iff it can be defined by a regular grammar.

1.1.2 Finite State Machines

The time required to determine if a string is in a given regular language is proportional to the length of the string, and the memory required has a constant bound unrelated to the length of the string.

We say that a regular language can be *recognized* by a *finite state machine* .

1.2 Context-Free Languages

A *context-free grammar* (or Chomsky type 2 grammar) is one where the left side of every production is an element of N. The language generated by a context-free grammar is called a *context-free language*.

Other, more general, grammars allow strings of terminals and nonterminal on the left side, so that a production can be used only in certain “contexts”. These more powerful grammars now are rarely, if ever, used when defining the syntax of a programming language. Instead, a superset of all valid programs is defined using a context-free grammar, and further restrictions are used to exclude “meaningless” programs.

The term *static semantics* is usually used for the collection of rules for excluding “meaningless” programs.

For example, the type correctness of a program is normally expressed by the static semantics of a language.

The static semantics of a language is often expresses as a function that returns a Boolean result.

We will limit our attention to the syntax of languages.

1.2.1 BNF

BNF is a commonly used notation for expressing the productions of a context-free language.

Notational Note: There are many different variations of *BNF*. In fact, there isn't even agreement on what "BNF" stands for.

Originally, "BNF" stood for *Backus Normal Form*, but now it is generally taken to stand for *Backus Naur Form*.

When *BNF* was introduced, programming language definitions were usually printed with a typewriter, not typesetting software.

The key feature of context-free grammars expressed using *BNF* is that productions with the same left side may be combined with alternative right sides separated by vertical bars.

Several less important conventions are used with traditional *BNF*.

- Nonterminals are contained inside angle brackets (because typewriters only have one font).
- The infix operator $::=$ is used instead of $\longrightarrow$ (because typewriters don't have a $\longrightarrow$ key).
- Terminal symbols are enclosed in quotation marks if necessary.

For example,

```
<digit>    ::= 0 | 1 | 2 | 3 | 4 | 5 |
               6 | 7 | 8 | 9
<integer>  ::= -<positive> | <positive>
<positive> ::= <digit> | <digit><positive>
```

Extended BNF, consider shortly, will allow us to simplify the above as follows.

```
<digit>    ::= 0 | 1 | 2 | 3 | 4 | 5 |
               6 | 7 | 8 | 9
<integer>  ::= [-]<digit>{<digit>}
```

1.2.2 Extended BNF

With *Extended BNF* ,

- Optional items can be enclosed in square brackets, [ ] .
- An item enclosed in curly brackets, { }, may be repeated, so that it occurs zero or more times.
- Ordinary parentheses, ( ) , can be used for grouping.
- The vertical bar, | , may be used within groupings.

For example,

```
<integer>  ::= [-|+]<digit>{<digit>}
```

indicates that an `<integer>` may or may not have a sign, while

```
<integer>  ::= (-|+)<digit>{<digit>}
```

indicates that an `<integer>` must have a sign.

In

```
<expression> ::=
    <identifier> |
    “(”<expression><operator><expression>“)”
```

the parentheses are in quote to make it clear that they are terminal symbols, not meta symbols used for grouping.

In modern usage, different fonts are usually used to distinguish terminal symbols, nonterminal symbols and meta symbols, making the use of angle brackets and quotes unnecessary.

Usually, each programming language definition uses its own unique variation of *BNF* when specifying the language syntax.

There is a *BNF* standard, somewhere, but by the time *BNF* was standardized it was too late to do much good. There is so much variation in *BNF* that there really isn't a default version, and I have never seen a document that claims to use standard *BNF*.

1.3 Using Grammars

There are several way to think informally about context-free grammars.

1.3.1 Language Generation

A context-free grammar can be regarded as a collection of recursive set equations.

These equations define the set of strings in a language, or generate the language.

A closely related way of viewing a context-free grammar is as a nondeterministic algorithm. (A regular grammar can be considered a special case of a context-free grammar.)

At each step during a derivation, we have two sorts of choices. We can decide which nonterminal to replace next, and which production to use in the replacement.

The choice of which nonterminal to replace isn't very important. In a context-free language, that nonterminal will be replaced sooner or later, and the set of possible productions to use in replacing that nonterminal will remain the same.

We can replace the nonterminals in any order, so for simplicity, let us assume that we will always replace the leftmost nonterminal. This results in a *leftmost derivation* .

With a context-free grammar, if a string is derivable from the start symbol then it is derivable using a leftmost derivation.

The choice of which production to use at each step of the leftmost derivation determines which string in the language will be generated by this nondeterministic process. Every string in the language, and only strings in the language, may be generated by this nondeterministic process. We note that, except in the very boring case of a finite language, the nondeterministic algorithm may fail to terminate.

(We make no assumptions about “fairness” or probabilities. We are interested only in what are the possible derivations.)

We can think of the nondeterministic algorithm in terms of a choice tree. Each node corresponds to a string over V. The root corresponds to the start symbol, S, as a string of length one, and the children of each node correspond to the strings that can be generated by replacing the leftmost nonterminal in the parent by each of the possible right sides of productions having that nonterminal on the left.

The leaves of this (usually infinite) tree are the strings in the language. The path from the root to a leaf indicates the sequence of choices made when generating the string at the leaf.

(Don’t confuse this with a parse tree, which is a finite tree corresponding to a single string in the language.)

1.3.2 Recognition

A language recognizer is an algorithm that determines whether a string is in a particular language.

Essentially, language recognizers work by seeing if a string can be generated by the grammar for the language. In effect, a recognizer must search the tree described on the previous page for a leaf corresponding to the string in question.

Of course, for this to be done reasonably efficiently, there must be a lot of pruning. Only a relative small portion of the tree should need to be explored.

2 Parsing

Parsing is similar to recognition, except we want to know the sequence of choices that would need to be made to generate a given string using a fixed grammar.

There is another way to look at this. When we write a program, we are not really thinking of a string of characters. We have more structure in mind. What we really have in mind is closer to being a tree than a string.

For example, a loop has several parts, including the loop body. Each of these parts may be decomposed into further parts, and so on.

Even a tree structure doesn't fully capture what we have in mind when we think about a program. There are many relationships that are not reflected in a hierarchical decomposition of a program into its component. For example, the requirement that the use of a variable is consistent with its declared type is not captured by a tree structure.

It is possible to define a context-sensitive grammar that will capture additional information such as this. For example, the programming language *Algol 68* has its syntax defined using a context-sensitive grammar. The grammar specifies fully what is and what is not a valid program.

In practice, it has been found to be far simpler to define programming languages with context-free grammars. The cost of this is that the language generated by the grammar includes all legal programs and a number of illegal ones as well.

As mentioned earlier, a "static semantics" is used to separate the valid and invalid ("meaningless") programs that are generated by the context-free grammar.

We will restrict our attention to the tree structure implied by the context-free grammar, for now.

The tree corresponding to a string in a language is called the *parse tree* of the string. Each internal node of a parse tree contains a nonterminal symbol and each leaf contains a terminal.

A context-free grammar more or less describes two things.

1. The structure of parse trees.
2. The mapping from a string representation of a program (or string in a language) to a corresponding parse tree.

In theory, the grammar should also tell us how to go from a parse tree back to the original string. In practice, we tend to throw out a lot of unimportant information, such as details about the white space, when constructing a parse tree.

2.1 Lexical and Context-Free Syntax

Usually, the parsing of computer programs is divided into two stages. (In practice, these stages may be intertwined with each other, and with other processes that conceptually occur later in the compilation of a program.)

Often, the description of a programming language is divided into two parts in a similar manner, where the lexical syntax groups symbols into tokens and the context-free syntax is defined over these tokens rather than individual characters.

With other languages, this distinction is not made in the formal definition of the language, but this division is reflected in the fact that the simplest object in the language, such as identifiers and constants, have a regular syntax, even if it is expressed in the form of a context-free grammar.

2.1.1 Lexical Syntax

The *lexical analysis* or *scanning* of a program breaks it into a sequence of tokens. For example, a sequence of letters and digits may be transformed into a single token representing an identifier. Similarly, numbers of various types are tokens. Some tokens may correspond to individual symbols in the original string. For example, the character + may generate a single token. Even in this case, the resulting token has been recognized as an operator and the token will normally carry this information.

Each type of token is defined by a regular language. The lexical analysis of a program basically simulates a finite state machine.

The *maximal munch* rule, which is used in the lexical definition of most programming languages, says that the longest possible initial portion of the unread input should be incorporated into the next token.

For example, "`casement `" would be recognized as an identifier, "`casement`" followed by white space, rather than the keyword `case` followed by the identifier "`ment`" (or eight one letter identifiers).

The need for the maximal munch rule (sometimes called the *principal of the longest substring*) arises because we are recognizing a sequence of tokens, one after another, in a single string.

The finite state machine often will need to consider several possible types of tokens at the same time. For example, if the first three characters in the token currently being processed are "`cas`", then the state of the finites state machine should reflect the fact that these may be the first three characters of an identifier, or the first three letters of the reserved word `case`.

With this two stage view, of scanning followed by *syntactic analysis* , the scanning or lexical analysis stage would not distinguish between different sorts of things where the difference is not reflected in the sequence of characters comprising the token.

For example, Haskell identifiers beginning with upper case letters are distinguished from identifiers beginning with lower case letters. The names of `constructors`, `type constructors`, `type classes` and `modules` all begin with upper case letters and otherwise have the same syntax. These four different sorts of identifiers are distinguished by context, but would be regarded as having tokens of the same type by a scanner.

The description of the syntax of a language may not fully reflect the division between lexical analysis and syntactic analysis. For example, the *Haskell Lexical Syntax*, given in section 9.2 of the *Haskell Report*, contains the productions

$$\begin{array}{rcl} \texttt{tycons} & \rightarrow & \texttt{conid} \\ \texttt{tycls} & \rightarrow & \texttt{conid} \\ \texttt{monid} & \rightarrow & \texttt{conid} \end{array}$$

reflecting the fact that *constructors*, *type constructors*, *type classes* and *modules* all have the same form and can be distinguished only by context. We would regard this differentiation as part of the context-free syntax.

2.1.2 Context-Free Syntax

The context-free syntax of a programming language defines a language over the tokens.

In theory, all identifiers of a particular type are identical. Similarly, all floating point constants are identical. Otherwise, we would not have a finite vocabulary as required by the definition of a phrase-structured grammar.

Of course, the actual names of identifiers, or at least sufficient information to distinguish identical identifiers from different identifiers, must be retained for later stages in compilation.

For example, type checking must be able to associate a declaration with each use of a variable (if variables must be declared). Even more important, the contents of different variables must be stored in different locations!

2.2 Parse Trees and Abstract Syntax Trees

A *parse tree* reflects how a string was parsed.

Each internal node of a parse tree is labeled with a nonterminal and each leaf node is labeled with a terminal. A parse tree will have the start symbol at its root node. Concatenating the leaves from left to right will give the string that was parsed.

When we parse a program, we determine a derivation of the program. In the derivation, each nonterminal is replaced, at some step, with a string over V. The subtrees of an internal node are labeled by the symbols from the right side of the production that was applied to the label of the parent in the derivation.

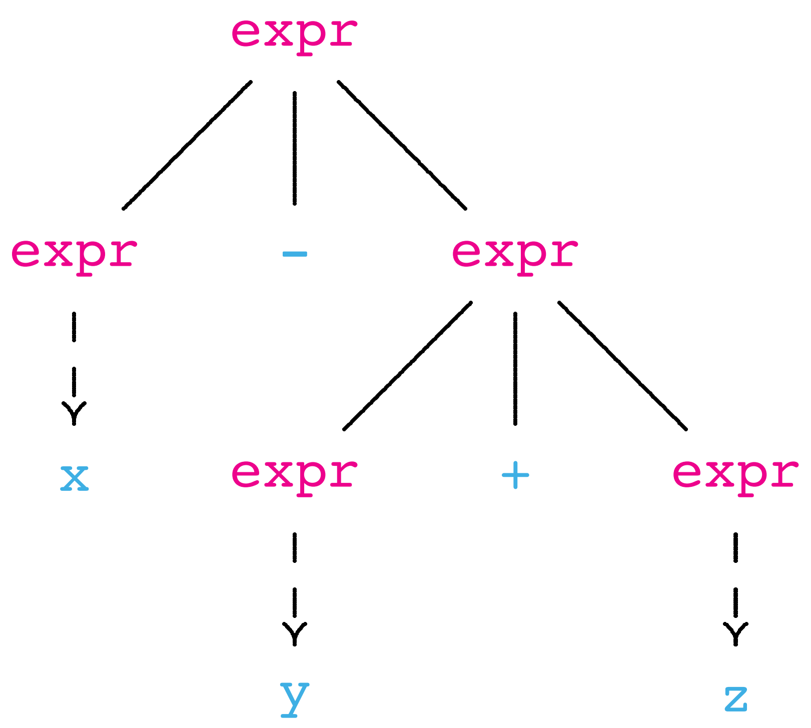
In practice, much of the information in a full parse tree can be discarded.

For example, with a production

$$\texttt{if-statement} \rightarrow$$
$$\texttt{if expression then statement else statement}$$

we know that an `if-statement` always generates an `if` and a `then` and an `else`. It is simpler to leave these out of the tree completely.

Similarly, if we have an expression `x-(y+z)`, the parentheses are necessary in the original program text, but may be dropped from the resulting tree since the tree structure reflects this information.



A parse tree with this unnecessary information removed is called an *abstract syntax tree* (or just a syntax tree).

2.2.1 Abstract Syntax

Abstract syntax is syntax that corresponds to an abstract syntax tree rather than a parse tree. Abstract syntax may not relate well to a raw string of symbols. For example, there is no need for parentheses in an abstract syntax tree representation of an expression, but these are sometimes necessary with a textual expression.

Haskell datatypes are well suited for expressing abstract syntax.

2.3 Associativity and Precedence

Binary operators with different associativity and precedence are found in most programming languages. The grammar for the language usually indicates how expressions should be parsed.

The C++ definition has many different expressions to represent the precedence and associativity of operators. (See Section 1.4 of Appendix A of the *C++ Standard*.) (Actually, I have a working draft of the standard, dated December 1996, so the location of the grammar may have been moved in the final version of the standard.)

The syntax of Java, as given in Chapter 18 (Syntax) of *Java Language Specification, Second Edition* does not indicate the precedence of operators, but the grammar given in Chapter 15 (Expressions) does (see sections 15.17 and 15.18 for example).

In Section 9.5 of the *Haskell Report*, an indexing scheme is used to compactly represent a collection of productions.

2.4 Ambiguity

A grammar is said to be *ambiguous* if there exists at least one string that has two or more different parse trees (or equivalently, has at least two different leftmost derivations.)

For the most part, you want a grammar defining a language to be unambiguous. Occasionally, a grammar will be ambiguous, with an English language explanation as to how the ambiguity should be resolved.

For example, the lambda calculus is often defined by the grammar

$$
\begin{aligned}
\texttt{exp} \rightarrow\ & \texttt{constant} \\
|\ & \texttt{variable} \\
|\ & \texttt{exp exp} \\
|\ & \lambda\ \texttt{variable}\ .\ \texttt{exp} \\
|\ & (\texttt{exp})
\end{aligned}
$$

together with a couple of rules to disambiguate the grammar.

1. Function application is left associative.
2. The body of a λ-expression extends as far to the right as possible.

I would like to thank Tai Meng for many corrections and suggestions for improvements to these notes. Of course, the remaining errors are my fault.