Context-free Grammars

> *Grammar:*
> that part of the study of language which deals with
> form and structure of words (*morphology*)
> and with their customary arrangement in
> phrases and sentences (*syntax*),
> usually distinguished from
> the study of word meanings (*semantics*).
> *−Webster*

Phrase Structure

Each programming language has some *symbols* reserved for fixed purposes. In addition there are classes of symbols such as identifiers and literal constants which the programmer can use to name higher level constructs of a program. The input text must be a sequence of these symbols, interspered with white space and comments. Each programming language also uses *phrase names* to define its syntax.

A *reduction* defines a phrase as a sequence of phrase names and symbols. For English, the rule

sentence ←subject predicate .

indicates that a subject followed by a predicate followed by a period is one allowed form of a sentence. There are usually several reductions defining any one phrase name. A reduction is a substitution rule. Wherever a contiguous exact match to the right-hand side of such a rule is found in a text, the matched text can be removed and replaced by the phrase name on the left side of the rule. The act of substitution is called reduction. The meta-symbol '←' is called the *reduction arrow.*

A *grammar* is a set of related reductions. One (or more) of the phrase names is the *goal* of the grammar. In English the goal is *sentence.* In programming the

goal is usually *program*. If, by a sequence of substitutions, the input text can be reduced to a goal, the input is syntactically correct.

For example, the propositional calculus provides combinations of boolean values. Table 1 presents a grammar for propositions. There are ten reductions. The symbol t represents true, f represents false.

$$
\begin{aligned}
Proposition &\leftarrow Disjunction \\
Disjunction &\leftarrow Disjunction \lor Conjunction \\
Disjunction &\leftarrow Conjunction \\
Conjunction &\leftarrow Conjunction \land Negation \\
Conjunction &\leftarrow Negation \\
Negation &\leftarrow \lnot\ Boolean \\
Negation &\leftarrow Boolean \\
Boolean &\leftarrow \text{t} \\
Boolean &\leftarrow \text{f} \\
Boolean &\leftarrow (\ Disjunction\ )
\end{aligned}
$$

Table 1: Proposition Grammar (mathematical form)

A sequence of reductions is called the *parse*. Suppose the input text is f∨t∧¬f. Here is a sequence of strings, each of which results from the former by the application of one substitution rule.

f ∨ t ∧¬ f
Boolean ∨ t ∧¬ f
Negation ∨ t ∧¬ f
Conjunction ∨ t ∧¬ f
Disjunction ∨ t ∧¬ f
Disjunction ∨ *Boolean* ∧¬ f
Disjunction ∨ *Negation* ∧¬ f
Disjunction ∨ *Conjunction* ∧¬ f
Disjunction ∨ *Conjunction* ∧¬ *Boolean*
Disjunction ∨ *Conjunction* ∧ *Negation*
Disjunction ∨ *Conjunction*
Disjunction
Proposition

The grammar allows many different choices at each step. Some of them are just a matter of order. The left-to-right order was chosen here because that is the way a parser will do it. Other choices were more serious. If the wrong choice had been made the parse would have gotten stuck without reaching its goal *Proposition*. For example, the first time *Disjunction* appeared, it could have immediately been rewritten as *Proposition*. See if you can find a few more places where the right decision was magically made. How to make the right choice (efficiently) is what automatic parsing is all about.

The parse can be displayed as a *syntax tree*, with the input at the leaves and the goal at the root. There are six symbols in the text, therefore one expects six leaves on the syntax tree. The tree is shown in Figure 1.
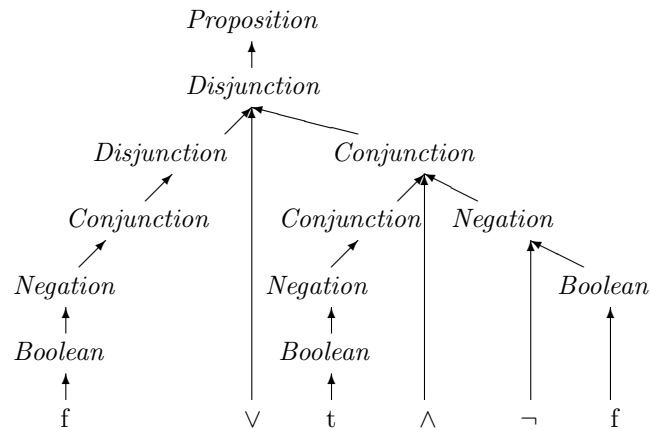
Figure 1
Syntax Tree for f ∨ t ∧ ¬f

One can express the same tree in linear functional notation, with the phrase names as functions and the subnodes as arguments.

$Proposition(Disjunction(Disjunction(Conjunction(Negation(Boolean(\text{f})))), \vee,$
$Conjunction(Conjunction(Negation(Boolean(\text{t}))), \wedge, Negation(\neg, Boolean(\text{f})))))$

Everyday Grammars for Programming Languages

In everyday use, for programming languages, it is more convenient to use page layout to present the reductions. All the rules defining a single phrase name are grouped together; the right sides of the reductions are indented; there is no need for the meta-symbol ←. (For the moment, consider the rule names as comments.)

CFG rule	rule name
Proposition	
Disjunction	*r0*
Disjunction	
Disjunction ∨ *Conjunction*	*r1*
Conjunction	*r2*
Conjunction	
Conjunction ∧ *Negation*	*r3*
Negation	*r4*
Negation	
¬ *Boolean*	*r5*
Boolean	*r6*

Boolean	
t	*r7*
f	*r8*
(*Disjunction*)	*r9*

Table 2: Everyday Grammar for Propositions

Exercises

(a) Build syntax trees for t, ((t)), and t∧f∧f.

(b) Relabel the nodes of the syntax tree with the rule names instead of the phrase names. For example, the top node is named *r0* instead of *Proposition*. This is the form of tree most often used in compilers.

Formal Definition of Context-free Grammars

Grammars like this are called *context-free grammars* (abbreviated as CFG), indicating that a phrase can be reduced independent of the symbols to its left and right in the text. This section presents a formal definition. The notation is largely conventional although the reader may wish to turn to the section on Notation for details and some concepts peculiar to this book. Formally, the 4-tuple

$$\mathcal{G} \stackrel{\text{def}}{=} \langle V_I, V_N, V_G, \Pi \rangle \tag{2}$$

consisting of a set of symbols, a set of phrase names, a set of goals, and a set of reductions, is a CFG if it satisfies the following constraints:

$$
\begin{aligned}
V &\stackrel{\text{def}}{=} V_I \cup V_N \\
V_I \cap V_N &\stackrel{\text{def}}{=} \{\} \\
V_G &\stackrel{\text{def}}{\subseteq} V_N \\
\Pi &\stackrel{\text{def}}{\subseteq} V_N \times V^*
\end{aligned}
$$

It is convenient to restrict the use of lower case Greek letters to represent sequences of symbols (strings). In particular, λ is used to represent the null string.

The members of Π are pairs: each consisting of a phrase name on the left and a phrase on the right. It is more mnemonic to write $p \leftarrow \alpha$ instead of $\langle p, \alpha \rangle$ for members of Π even in the middle of formulas. The arrow notation is consistent with the fact that $\leftarrow$ is a relation.

For the example proposition grammar, the sets are:

$$
\begin{aligned}
V_N &= \{Proposition, Disjunction, Conjunction, Negation, Boolean\} \\
V_I &= \{\vee, \wedge, \neg, \mathrm{t}, \mathrm{f}, (,)\} \\
V_G &= \{Proposition\}
\end{aligned}
$$

Usually one first writes down Π and infers V_I, V_N, V_G and, perhaps, a unique goal symbol G. The phrase names are the symbols used on the left of the rules of Π (sometimes called the *nonterminal vocabulary*). The right sides of the rules use in addition all of the input symbols (sometimes called the *terminal vocabulary*). By convention, the first rule of Π defines a goal G and goals are used only on the left of rules.

Some new notation is required. If R is a set of pairs, $\mathcal{D}(R)$ is the domain of R and $\mathcal{R}(R)$ is the range of R. If B is a set of sequences of symbols, then $B^{1/*}$ is the set of symbols found in any sequence in B. In particular, $(A^*)^{1/*} = A$. $choice(S)$ is one element of S. Thus $S \neq \{\} \Rightarrow choice(S) \in S$.

$$
\begin{aligned}
V_N &\overset{\text{def}}{=} \mathcal{D}(\Pi) \\
V_I &\overset{\text{def}}{=} \mathcal{R}(\Pi)^{1/*} - V_N \\
V_G &\overset{\text{def}}{=} V - \mathcal{R}(\Pi)^{1/*} \\
G &\overset{\text{def}}{=} \text{choice}(V_G)
\end{aligned}
$$

Definition of Language

The finite relation $\leftarrow$ on $V_N \times V^*$ can be extended to $V^* \times V^*$ by allowing constant context. That is,

$$\alpha \in V^* \wedge \gamma \in V^* \wedge B \leftarrow \beta \Rightarrow \alpha B \gamma \leftarrow \alpha \beta \gamma$$

Moreover, the relation $\leftarrow$ is transitive on $V^* \times V^*$, so

$$\beta \leftarrow^* \alpha$$

means there is a sequence of zero or more rewrites reducing α to β. If the sequence of rewriting rules is ρ, then

$$\beta \overset{\rho}{\leftarrow} \alpha$$

means the same thing.

The *language* defined by a grammar $\mathcal{G}$ is the set of strings of input symbols that can be reduced to a goal.

$$\mathcal{L}(\mathcal{G}) \stackrel{\text{def}}{=} \{\alpha \mid G \leftarrow^* \alpha) \wedge G \in V_G\} \cap V_I^*$$

Since, as noted earlier, we can derive $\mathcal{G}$ from Π, it is also reasonable to write $\mathcal{L}(\Pi)$ to mean the same thing as $\mathcal{L}(\mathcal{G})$.

Left-to-right Application

The definition of language does not specify the order in which the reductions are to be applied. In fact processing is always left-to-right for a number of reasons. There is a *parse stack* σ and *input text* τ. Input symbols are stacked until a phrase is on the top of the stack. Then the phrase is popped off and replaced with its phrase name. This continues until all the input is consumed and the parse stack contains only a goal symbol.

Left-to-right order is required by the following alternative definition of language. There is a predicate $\mathcal{C}$ which can be used to prove a string is in the language defined by the grammar.

$$
\begin{aligned}
G \in V_G &\Rightarrow \mathcal{C}(G, \lambda) \\
B \leftarrow \beta \in \Pi \wedge \mathcal{C}(\sigma B, \tau) &\Rightarrow \mathcal{C}(\sigma \beta, \tau) \\
a \in V_I \wedge \mathcal{C}(\sigma a, \tau) &\Rightarrow \mathcal{C}(\sigma, a\tau) \\
\mathcal{C}(\lambda, \tau) &\Rightarrow \tau \in \mathcal{L}(\mathcal{G})
\end{aligned}
$$

Table 3: Left-to-right Reduction Application

The first implication defines the goal. The second implication applies a reduction on the top of the parse stack. The third implication shifts a symbol from the input text and pushes it on the top of the stack. The *shift/reduce sequence* is the reverse of the steps of the proof. Repeating the parse of f $\vee$ t $\wedge$ ¬f:

predicate	*rule*
$\mathcal{C}(\overline{Proposition}, \lambda)$	
$\mathcal{C}(Disjunction, \lambda)$	$r0$
$\mathcal{C}(Disjunction \vee Conjunction, \lambda)$	$r1$
$\mathcal{C}(Disjunction \vee Conjunction \wedge Negation, \lambda)$	$r3$
$\mathcal{C}(Disjunction \vee Conjunction \wedge \neg Boolean, \lambda)$	$r5$
$\mathcal{C}(Disjunction \vee Conjunction \wedge \neg f, \lambda)$	$r8$
$\mathcal{C}(Disjunction \vee Conjunction \wedge \neg, f)$	$shift$ f

$$\mathcal{C}(Disjunction \vee Conjunction\wedge, \neg\mathrm{f}) \qquad shift \; \neg$$
$$\mathcal{C}(Disjunction \vee Conjunction, \wedge\neg\mathrm{f}) \qquad shift \; \wedge$$
$$\mathcal{C}(Disjunction \vee Negation, \wedge\neg\mathrm{f}) \qquad r4$$
$$\mathcal{C}(Disjunction \vee Boolean, \wedge\neg\mathrm{f}) \qquad r6$$
$$\mathcal{C}(Disjunction \vee \mathrm{t}, \wedge\neg\mathrm{f}) \qquad r7$$
$$\mathcal{C}(Disjunction\vee, \mathrm{t} \wedge \neg\mathrm{f}) \qquad shift \; \mathrm{t}$$
$$\mathcal{C}(Disjunction, \vee\mathrm{t} \wedge \neg\mathrm{f}) \qquad shift \; \vee$$
$$\mathcal{C}(Conjunction, \vee\mathrm{t} \wedge \neg\mathrm{f}) \qquad r2$$
$$\mathcal{C}(Negation, \vee\mathrm{t} \wedge \neg\mathrm{f}) \qquad r4$$
$$\mathcal{C}(Boolean, \vee\mathrm{t} \wedge \neg\mathrm{f}) \qquad r6$$
$$\mathcal{C}(\mathrm{f}, \vee\mathrm{t} \wedge \neg\mathrm{f}) \qquad r8$$
$$\mathcal{C}(\lambda, \mathrm{f} \vee \mathrm{t} \wedge \neg\mathrm{f}) \qquad shift \; \mathrm{f}$$

Table 4: Proof of, and shift/reduce sequence for, 'f $\vee$ t $\wedge \neg$f' $\in \mathcal{L}(\mathcal{G})$

The *parsing problem* is finding a shift/reduce sequence given the CFG and an input text.

CFG **defects**

As it turns out, a particular CFG may be defective. Having extraneous symbols in the CFG is one form of defect. Another is having a property that will defeat some tool that is going to be used.

Some new notation is useful. For a relation R and set S, the *domain restriction* of R to S is $S \triangleleft R \stackrel{\mathrm{def}}{=} (S \times \overline{\{\}}) \cap R$ and the *range restriction* of R to S is $R \triangleright S \stackrel{\mathrm{def}}{=} R \cap (\overline{\{\}} \times S)$.

The following table lists some properties which may, or may not, be considered requirements. For all A in V_N:

$size(V_G) = 1$	unique goal
$\Pi \triangleright \{\lambda\} = \{\}$	no erasure
$A \not\leftarrow^+ A\alpha$	no left recursion
$A \not\leftarrow^+ A$	no circularity
$\{\alpha \mid A \leftarrow^* \alpha\} \cap V_I^* \neq \{\}$	no useless phrase names
$\mathcal{L}(\mathcal{G})^{1/*} = V_I$	no useless symbols in V

Table 5: Possible Restrictions on CFGs

Transforming CFGs

A grammar can be transformed by treating rules as text. A reduction can be applied to the right hand side of another reduction or, inversely, a phrase can be substituted for its phrase name. Because a CFG allows

context-free substitution, one can see that no new strings are introduced into the language by either of these kinds of transformations.

Let Π be a given set of rules. Then for the transformations $\mathcal{T}$ below, $\mathcal{L}(\mathcal{T}(\Pi)) = \mathcal{L}(\Pi)$. Doing a transformation may introduce other opportunites for transformations to clean up Π. Sometimes a new symbol $X \notin V_N$ must be introduced.

transform	*condition*	$\mathcal{T}(\Pi)$
substitute	$A \leftarrow \alpha B \gamma \in \Pi \wedge B \leftarrow \beta \in \Pi$	$\Pi \cup \{A \leftarrow \alpha \beta \gamma\}$
circular	$A \leftarrow A \in \Pi$	$\Pi - \{A \leftarrow A\}$
useless	$A \leftarrow \alpha \in \Pi \wedge A \notin V_G \cap \mathcal{R}(\Pi)^{1/*}$	$\Pi - \{A \leftarrow \alpha\}$
shorten rule	$A \leftarrow \beta b \in \Pi$	$\Pi - \{A \leftarrow \beta b\} \cup \{A \leftarrow Xb, X \leftarrow \beta\}$
fuse rules	$A \leftarrow \beta \in \Pi \wedge B \leftarrow \beta \in \Pi$	see fusion discussion below
remove rule	$A \leftarrow \beta \in \Pi$	see removal discussion below

Table 6: CFG Transformations

Fusion

Fusion will be significant for the discussion of finite automata. The object is to combine rules with different phrase names but the same phrase definition. The first step is to add a new rule replacing both of the rules with the identical right hand sides.

$$\Pi \cup \{X \leftarrow \beta\} - \{A \leftarrow \beta, B \leftarrow \beta\}$$

Then each rule in Π using A or B on the right hand side must be duplicated, with A or B replaced by X. That is, for each $C \leftarrow \alpha A \gamma \in \Pi$

$$\Pi = \Pi \cup \{C \leftarrow \alpha X \gamma\}$$

and similiarly for B. There is an obvious generalization for three or more identical right hand sides.

Removal

Any rule can be removed from the grammar by systematically back substituting its right-hand-side for its phrase name. The new Π first substitutes β for A in all rules $B \leftarrow \alpha A \gamma$, adding the new rules to the CFG,

$$\Pi = \Pi \cup \{B \leftarrow \alpha \beta \gamma\}$$

then removes the rule itself

$$\Pi = \Pi \cup \{A \leftarrow \beta\}$$

Erasure Removal

When β is λ, we have erasure. Avoiding erasure is a restriction of convenience for many grammar-oriented tasks; on the other other hand, erasure is useful for conciseness and representing semantics. The problem with erasure for parsers is that a phrase name can be created out of nothing. This makes parsing decisions at least locally nondeterministic.

The transformation above might create an erasing rule, so an iteration is required to clean all erasure[1] out of the CFG. The following iteration for an equivalent erasure-free set of rules Π' converges rapidly.

$$\Pi_0 = \Pi$$
$$\Pi_{i+1} = \Pi_i \cup \{B{\leftarrow}\alpha\gamma \mid A{\leftarrow}\lambda \in \Pi_i \wedge B{\leftarrow}\alpha A\gamma \in \Pi_i\}$$
$$\Pi_\infty = \overset{\lim}{\scriptstyle i\to\infty} \Pi_i$$

$$\Pi' = (\Pi_\infty \triangleright \overline{\{\lambda\}}) \cup (V_G \triangleleft \Pi_\infty)$$

Table 7: λ Removal

Substitute

Suppose one systematically back-substitutes the grammar for propositions, starting with *Boolean*. After the first stage there are six rules for *Negation* and the phrase name *Boolean* has been eliminated. The new grammar describes the same language as before.

$$
\begin{aligned}
\textit{Proposition} &\leftarrow \textit{Disjunction} \\
\textit{Disjunction} &\leftarrow \textit{Disjunction} \vee \textit{Conjunction} \\
\textit{Disjunction} &\leftarrow \textit{Conjunction} \\
\textit{Conjunction} &\leftarrow \textit{Conjunction} \wedge \textit{Negation} \\
\textit{Conjunction} &\leftarrow \textit{Negation} \\
\textit{Negation} &\leftarrow \neg \; \mathrm{t} \\
\textit{Negation} &\leftarrow \neg \; \mathrm{f} \\
\textit{Negation} &\leftarrow \neg \; (\; \textit{Disjunction}\;) \\
\textit{Negation} &\leftarrow \mathrm{t} \\
\textit{Negation} &\leftarrow \mathrm{f} \\
\textit{Negation} &\leftarrow (\; \textit{Disjunction}\;)
\end{aligned}
$$

So far one phrase name has been eliminated and the size of the rule set increased by one. One can continue in this way to remove *Negation* and *Conjunction*, ending with more rules and only two phrase names. The reader can form an opinion as to which grammar is preferable. The technique will have many uses.

---

[1] ... unless the empty string itself is in the language.

Alternative Notations

> The syntax will be described
> with the aid of metalinguistic formulae.
> Their interpretation is best explained by an example.
> — *Report on Algol 60*

There are other notations for grammars. In all of them there are *meta-symbols* which must be distinguished from the input symbols and the phrase names. Since white space is rarely significant in programming languages, the everyday grammar (cleverly) uses white space for the meta-symbols, thus avoiding the conflict. In general, this trick is insufficient.

The earliest popular form of context-free grammar, called BNF, appeared in the definition of ALGOL 60. BNF used the compound symbol '::=' where ← is used above; phrase names were bracketed; the reserved words were bold face; and the symbol | was used to separate alternative definitions for a single phrase name. Here is an excerpt:

⟨Boolean secondary⟩ ::= ⟨Boolean primary⟩ | ¬ ⟨Boolean primary⟩
⟨Boolean factor⟩ ::= ⟨Boolean secondary⟩ | ⟨Boolean factor⟩ ∧ ⟨Boolean secondary⟩
⟨Boolean term⟩ ::= ⟨Boolean factor⟩ | ⟨Boolean term⟩ ∨ ⟨Boolean factor⟩

The ISO C Standard uses a notation similar to the everyday notation. Phrase names are in italic font, may be hyphenated, and are followed by a ':' when defining a rule. The input symbols are set in mono-width font. Two extensions are used, 'one of' introduces a list of phrases, and $_{opt}$ follows an optional item in a phrase. Here is an excerpt:

logical-AND-expression:
       inclusive-OR-expression
       logical-AND-expression `&&` *inclusive-OR-expression*

logical-OR-expression:
       logical-AND-expression
       logical-OR-expression `||` *logical-AND-expression*

Another practical grammar notation arises from the need to provide meta-symbols and also convenient computer input. The previous trick of using white space does not generalize, for instance to using grammars to describe grammars themselves, or when white space must be described or when rules are too long to place on one line, or when typesetting is not convenient. The requirement that $V_I \cap V_N = \{\}$ prevents grammars from describing elements of V_N.

One solution is to ignore white space, to quote input symbols with ' '', use identifiers for phrase names, and leave everything else for meta-use.

The result of these changes is a free-form notation similar to that for modern programming languages. The meta-symbol indicating "is defined as" becomes '='; left-hand sides are repeated as often as necessary; "end of rule" is indicated by ';'.

```
Proposition = Disjunction ;
Disjunction = Disjunction '∨' Conjunction ;
Disjunction = Conjunction ;
Conjunction = Conjunction '∧' Negation ;
Conjunction = Negation ;
Negation    = '¬' Boolean ;
Negation    = Boolean ;
Boolean     = 't';
Boolean     = 'f';
Boolean     = '(' Disjunction ')';
```

Table 8: Free-form Proposition CFG

The notation in Table 2 allowed V_N and V_I to be inferred from a concrete representation of Π. The inference required global information: one could not know if a symbol was an input or a phrase name until the entire set of rules had been examined (t and f, for example, are input symbols only because they fail to appear on the left of a defining rule).

In the new notation each rule stands by itself and inputs can be immediately distinguished from phrase names. There are three special space-saving phrase names — Letter, Digit and Any — which are grammatical builtins meaning any letter, any digit and any (printable) character.

Self-describing CFG

Ignoring white space, the new notation can be used to describe itself as in Table 9.

```
Grammar =  Rules;
Rules   =  Rules Rule;
Rules   =  ;
Rule    =  Name '=' Phrase ';';
Phrase  =  Phrase Symbol;
Phrase  =  ;
Symbol  =  Name;
Symbol  =  Input;
Name    =  Letter;
Name    =  Name Letter;
Name    =  Name Digit;
```

```
Input   =  ''' Any ''';
```

Table 9: A Self-describing CFG

Exercises

(a) What is V_I in the grammar-grammar in Table 9?

(b) Use the CFG in Table 9 to parse itself. Actually, it is enough just to parse the rule that describes itself — leave big parses to the computer.