Context-Free Languages

- A formalism such as, regular expressions, can be viewed as a *language generator*.
- We will now look at a more complex kind of language generator: *context-free grammars* (*CFGs*)
- CFG's use *variables* to represent strings of symbols, also called *terminals*.
- For each variable, there is a collection of *rules* or *productions* which determine the strings that it represents.
- There is one special variable, the *start variable*.

Example

Suppose S represents strings given by the regular expression ab^*ab^*. Then S has the form $aBaB$, where B represents strings given b^*. This is expressed by the following rule:

$$S \rightarrow aBaB$$

How can we define B?

1. B can be the empty string, or

2. B can be a string starting with b, followed by a string in b^*

This can be expressed by the following rules:

$$\begin{aligned} B &\rightarrow \epsilon \\ B &\rightarrow bB \end{aligned}$$

Deriving Strings

Using these rules, we may derive strings by starting with a start symbol S, and repeatedly replacing one nonterminal with the right hand side of a rule for that nonterminal, until we reach a string with no nonterminals, e.g., $abba$ is obtained via the following sequence:

1. S
2. $aBaB$ (since $S \rightarrow aBaB$)
3. $abBaB$ (since $B \rightarrow bB$)
4. $abbBaB$ (since $B \rightarrow bB$)
5. $abbaB$ (since $B \rightarrow \epsilon$)
6. $abba$ (since $B \rightarrow \epsilon$)

Formal Definition of CFG's

A *context-free grammar* is a 4-tuple $G = (V, \Sigma, R, S)$, where

- V is the set of *variables*;
- Σ is the set of *terminals*;
- R is a set of *rules* of the form $A \to \alpha$ where A is a variable $\to$ is the *production symbol* and α is a string of 0 or more terminals and variables called the *body* of the production.
- $S \in V$ is the *start variable*.

Formal Definition of Derivation in CFG's

Suppose α and β are strings in $(V \cup \Sigma)^*$. Let A be a variable, and $A \rightarrow \gamma$ be a production of G. Then we write $\alpha A \beta \Rightarrow_G \alpha\gamma\beta$ and say $\alpha\gamma\beta$ is *derivable from* $\alpha A \beta$, *in one step*.

In the example above, $aBaB \Rightarrow abBaB$.

Note: the term *context-free* comes from the fact that we can apply the rule $A \rightarrow \gamma$, regardless of what α and β might be.

$\Rightarrow_G^*$ denotes derviability in zero or more steps.

A sequence of the form:

$$w_0 \Rightarrow_G w_1 \Rightarrow_G \cdots \Rightarrow_G w_n$$

is called a derivation (with n steps).

Context-Free Languages

$L(G)$, the *language of G*, is defined by:

$$L(G) = \{w \in \Sigma^* | S \Rightarrow_G^* w\},$$

i.e., $L(G)$ is the set of all strings of terminals derivable from S.

A language is *context-free* if it is of the form $L(G)$ for some CFG G.

Applications

- Specifying the syntax of a programming language. E.g., YACC is a mechanical way of turning a (suitably restricted) CFG into a parser.
- Document Type Definition (DTD) in XML. XML is a language used to add tags to a document to describe its semantics. People who want to use XML set up their system of tags using DTD, which is essentially a context-free grammar.

Arithmetic Expressions

For example, as part of a programming language we might want to specify all strings strings over $\{+, *, (,), \mathsf{num}, \mathsf{id}\}$ which represent legal arithmetic expressions.

Here num stands for any legal number, and id for any legal identifier. While these could be specified using the grammar as in the text example, it is normal practice to specify these *token classes* using regular expressions. (NOTE: In this language, id and num are meant to be single terminals.)

Examples of legal strings are:

$(\mathsf{num} + \mathsf{id}) * \mathsf{id}$
$\mathsf{num} * \mathsf{num} + \mathsf{num}$

Example

A grammar $G = (V, \Sigma, P, E)$ for such expressions is given by:

$$\begin{array}{rcl} V & = & \{E\}, \\ \Sigma & = & \{+, *, (,), \mathsf{id}, \mathsf{num}\}, \end{array}$$

where P has the following rules:

$$\begin{array}{rcl} E & \to & E + E \\ E & \to & E * E \\ E & \to & (E) \\ E & \to & \mathsf{id} \\ E & \to & \mathsf{num} \end{array}$$

Exercise: derive the following strings:

$(\text{num} + \text{id}) * \text{id}$

$\text{num} * \text{num} + \text{num}$

Context-Free and Regular Languages

There are languages which are context-free but not regular, e.g.,

1. $\{a^n b^n \mid n \geq 0\}$
2. $\{w \in \{a,b\}^* \mid w = w^R\}$
3. $\{w \in \{(,)\}^* \mid w \text{ is balanced}\}$

On the other hand, every regular language is context-free (we will show this later.)

Palindromes: $\{w \in \{0,1\}^* \mid w = w^R\}$

Let $G_{pal} = (\{S\}, \{0,1\}, P, S)$ be the grammar with the following set P of productions:

$$\begin{aligned} S &\rightarrow \epsilon \\ S &\rightarrow 0 \\ S &\rightarrow 1 \\ S &\rightarrow 0S0 \\ S &\rightarrow 1S1 \end{aligned}$$

$L(G_{pal})$ is the language of palandromes over $\{0, 1\}$

PROOF: We show that $w \in L(G_{pal})$ iff w is a palindrome, by induction on $|w|$.

(If)

BASIS: If $w = \epsilon, 0$ or 1 then it's derived by one of the first three production rules.

INDUCTION: If $|w| \geq 2$ then $w = 0x0$ or $w = 1x1$ for some string x which is also a palindrome. Then by the fourth or fifth production rule, we can apply either $S \rightarrow 0S0$ or $S \rightarrow 1S1$. By induction $S \Rightarrow^* x$, so we can derive w by applying one of these two rules, following by one or more productions to derive the middle of w.

Palindrome Proof (cont'd)

(Only if)

If w is in $L(G_{pal})$ then we want to show w is a palindrome. The proof is by induction on the number of steps used to derive w.

BASIS: One step. $w = \epsilon$, 0 or 1, so w is a palindrome.

INDUCTION: Suppose it takes $n + 1 > 1$ steps.

Then the first step in the derivation is $S \Rightarrow 0S0$ or $S \Rightarrow 1S1$.

The next n steps must be a derivation $S \Rightarrow^* x$ where $w = 0x0$ or $1x1$. By induction, x is a palindrome. Then $w = 0x0$ or $1x1$ is a palindrome.

Parse Trees

A string in $L(G)$ may have many different derivations.

Some derivations of the same string differ only in the order in which variables are expanded.

A *parse tree* is a way of representing derivations which ignores this kind of ordering.

Parse trees are very useful in compilation, since they more or less correspond to the data structures used to internally represent a source program

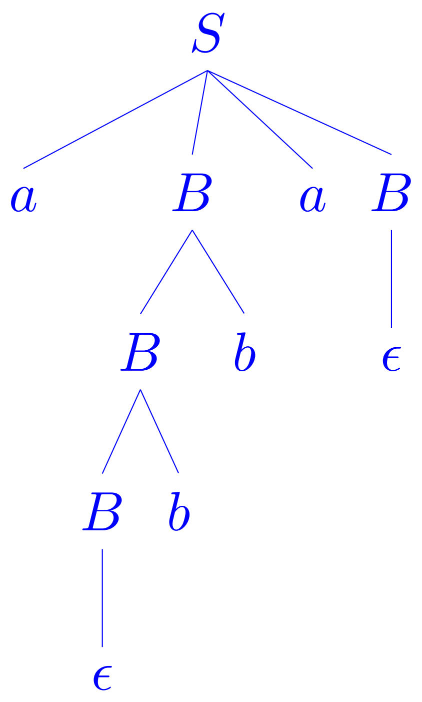
Example

Consider the grammar with rules $S \rightarrow aBaB$, $B \rightarrow \epsilon$, $B \rightarrow bB$. The following derivations of the string $abba$ have the same parse tree, shown below:

$$D_1 = S \Rightarrow aBaB \Rightarrow abBaB \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$$

$$D_2 = S \Rightarrow aBaB \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$$

Example



Parse Trees – Definition

Let $G = (V, \Sigma, P, S)$ be a CFG. A tree T is a *parse tree* for G if

- each internal node of T are labeled by a variable.
- each leaf is labeled by a terminal, ϵ, or a variable. If a leaf is labeled ϵ then it is the only child of its parent.
- if an internal node is labelled by A, and its children are labeled (in order) by $X_1, \ldots, X_k$, then $A \to X_1 \ldots X_k$ is a production in P.

Yield of a Parse Tree

The *yield* of a parse tree T is the string obtained by concatenating the labels of the leaves as they are visited left to right.

If

- the root is labelled by S
- the leaves of T are labeled by terminals or ϵ

then the yield is a string in the language of G.

Example

A parse tree for the string $(id + id) * (num + id)$ in the arithmetic expression language:

Left- and Rightmost Derivations

The *leftmost* derivation is the derivation in which the leftmost variable in the currently derived string is expanded.

Similarly, there is a *rightmost* derivation.

Standard parsing algorithms for compilers use either leftmost or rightmost derivations.

Ambiguous Grammars

In the grammar for arithmetic expressions, there are two possible parse trees for the string: $num + num * num$:

Ambiguous Grammars

- Grammars which can have more than one parse tree for the same string are *ambiguous*.
- Practically, this can cause problems, e.g. what if we are using parse trees as part of an algorithm to do expression evaluation?
- **Theorem:** For each grammar (V, Σ, P, S) and string $w \in \Sigma^*$, w has two distinct parse trees iff w has two distinct leftmost derivations from S.

Ambiguous Grammars

Fortunately, there is a version of the arithmetic expression grammar which is not ambiguous. The rules are: $E \to E+T$, $E \to T$, $T \to F$, $T \to T*F$, $F \to (E)$, $F \to num$, $F \to id$.

What is the parse tree for $num + num * num$ in this grammar?

There are *inherently ambiguous* grammars for which no equivalent unambiguous grammars exist. Such grammars should not be used for specify programming language syntax.

Every Regular Language is a CFL

We start with a DFA $M = (Q, \Sigma, \delta, q_0, F)$

1. Make a variable R_i for each $q_i \in Q$
2. Add the rule $R_i \rightarrow aR_j$ if $\delta(q_i, a) = q_j$.
3. Add the rule $R_i \rightarrow \epsilon$ if $q_i \in F$.
4. The start variable is R_0.
5. How to show this is correct?

From Regular Expressions to CFGs

Another way to see this.

Clearly $\emptyset$ is generated by a grammar with no productions, a is generated by the grammar with a single production $S \to a$, and ϵ is generated by $S \to \epsilon$.

Now suppose that R_1 is generated by $G_1 = (V_1, \Sigma, P_1, S_1)$ and R_2 is generated by $G_2 = (V_2, \Sigma, P_2, S_2)$.

- $R_1 \cup R_2$ is generated by $G = (V, \Sigma, P, S)$ where $V = V_1 \cup V_2 \cup \{S\}$ and $P = P_1 \cup P_2 \cup \{S \to S_1 \mid S_2\}$
- $R_1 \cdot R_2$ is generated by $G = (V, \Sigma, P, S)$ where $V = V_1 \cup V_2 \cup \{S\}$ and $P = P_1 \cup P_2 \cup \{S \to S_1 S_2\}$
- R_1^* is generated by $G = (V, \Sigma, P, S)$ where $V = V_1 \cup \{S\}$ and $P = P_1 \cup \{S \to \epsilon \mid S_1 S\}$

Chomsky Normal Form

- Every nonempty CFL has a grammar where all productions are of the form $A \rightarrow BC$ where A, B and C are variables or $A \rightarrow a$ where A is a variable and a is a terminal.

  In addition, we permit the rule $S \rightarrow \epsilon$.

Converting to CNF

Theorem: Any CFL is generated by a context-free grammar in Chomsky normal form.

Proof:

1. Add new start variable S_0, and the rule $S_0 \rightarrow S$. If $\epsilon \in L(G)$, add a rule $S_0 \rightarrow \epsilon$.

2. Remove all rules of the form $A \rightarrow \epsilon$, $A \neq S_0$ and for every occurence of A in a rule of the form $R \rightarrow uAv$ where u, v are any strings of variables and terminals (possibly containing A), add the rule $R \rightarrow uv$.

   Each rule may be replaced by several rules! E.g., $R \rightarrow uAvAw$ is replaced by $R \rightarrow uAvAw$, $R \rightarrow uvAw$, $R \rightarrow uAvw$ and $R \rightarrow uvw$.

3. Remove every unit rule $A \to B$, and for every rule of the form $B \to u$ (where u can be any string of variables and terminals,) add a rule $A \to u$ unless is a previously removed unit rule.

4. Each rule of the form $A \to u_1 u_2 \ldots u_k$, $k \geq 3$ where the u_i are terminal symbols or variables is relaced by

$$\begin{aligned} A &\to u_1 A_1 \\ A_1 &\to u_2 A_2 \\ &\ldots \\ A_{k-1} &\to u_{k-1} u_k \end{aligned}$$

   For any terminal u_i in a rule of the form $A \to u_1 u_2$, replace it with a new variable U_i, and add a rule $U_i \to u_i$.

The CYK Algorithm

The *Cocke*, *Younger*, *Kasami* algorithm for a grammar $G = (V, \Sigma, P, S)$ allows us to determine, whether $w \in L(G)$ in time $O(n^3)$, where $n = |w|$.

Assume that G is in CNF

Consider a $n \times n$ (upper triangular) table T, where for $i \leq j$,

$$T(i,j) = \{A \in V \mid A \Rightarrow_G^* w_i \dots w_j\}$$

I.e., all variables from which we can derive the substring of w from positions i to j

Once we have filled in this table, answer "yes" iff $S \in T(1,n)$.

Filling in the Table

Clearly, $A \in T(i,i)$ iff $A \to w_i$ is a rule of G (why?)

When $i < j$, $T(i,j)$ is filled in as follows: for $k = i, \ldots, j-1$, if $B \in T(i,k)$ and $C \in T(k+1,j)$ and $A \to BC$ is a rule of G, then add A to $T(i,j)$.

We first fill in entries for length 1 substrings (diagonals), followed by length 2 subtrings, etc.

There are $O(n^2)$ entries, and it takes time $O(n)$ to fill in each one.