Writing Grammar

- Capabilities of CFG
- Verifying the language generated by a grammar
- Eliminating ambiguity
- Elimination of left recursion
- Left factoring

Capabilities of CFG

- Every construct that can be described by a regular expression can also be described by a CFG
- (e.g)

(a|b)*abb $\Rightarrow$ $A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$

$A_1 \rightarrow bA_2$

$A_2 \rightarrow bA_3$

$A_3 \rightarrow \varepsilon$

- Check for the string aababb

Algorithm to construct NFA to grammar

For each state i of the NFA, create a non terminal Ai

Begin

    If state I has a transition to state j on symbol a

        Introduce production Ai ->aAj

    If state I goes to state j on input ε

        Introduce production Ai -> Aj

End

If I is an accepting state

    Introduce Ai -> ε

If I is the start state

    Make Ai be the start symbol for the grammar

Example

- For the states 0 to 3 of NFA create NTs A0 to A3
- For A0

  a : A0 -> aA0 , A0 -> aA1

  b : A0 -> bA0
- For A1

  b : A1 -> bA2
- For A2

  b : A2 -> bA3
- For A3(accepting state)

  A3 -> ε
- 0 is the start state for NFA, hence A0 is the start state for the grammar

Verifying the language generated by a grammar

1. We must show that every string generated by g is in L
2. Every string in L can be generated by G

- (e.g)

  $S \rightarrow (S)\ S \mid \varepsilon$

Verifying the language generated by a grammar (cont.)

1. Every string generated by S is balanced

   S -> ε (empty string, hence balanced)

   S -> (S) S

   -> (x) S

   -> (x) y

2. Every balanced strings are generated or derivable by S

(x)y

s -> (S) S

-> (x) S

-> (x)y

((x)y)

S -> (S) S

-> ((S)S)S

-> ((x)S)S

-> ((x)y)S

-> ((x)y)

Left Recursion and Left Factoring

Left Recursion

- A grammar is ***left recursive*** if it has a non-terminal A such that there is a derivation.

  $A \stackrel{+}{\Rightarrow} A\alpha$ for some string α

- Top-down parsing techniques **cannot** handle left-recursive grammars.
- So, we have to convert our left-recursive grammar into an equivalent grammar which is not left-recursive.
- The left-recursion may appear in a single step of the derivation (*immediate left-recursion*), or may appear in more than one step of the derivation.

Immediate Left-Recursion

$A \rightarrow A\,\alpha \mid \beta$ where β does not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta\, A'$

$A' \rightarrow \alpha\, A' \mid \varepsilon$ an equivalent grammar

In general,

$A \rightarrow A\,\alpha_1 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ where $\beta_1 \ldots \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta_1\, A' \mid \ldots \mid \beta_n\, A'$

$A' \rightarrow \alpha_1\, A' \mid \ldots \mid \alpha_m\, A' \mid \varepsilon$ an equivalent grammar

Example

$$E \rightarrow E+T \mid T$$

$$T \rightarrow T*F \mid F$$

$$F \rightarrow id \mid (E)$$

$\Downarrow$ eliminate immediate left recursion

$$E \rightarrow T E'$$

$$E' \rightarrow +T E' \mid \varepsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow *F T' \mid \varepsilon$$

$$F \rightarrow id \mid (E)$$

Left-Recursion -- Problem

- A grammar cannot be immediately left-recursive, but it still can be left-recursive.
- By just eliminating the immediate left-recursion, we may not get a grammar which is not left-recursive.

$S \rightarrow Aa \mid b$

$A \rightarrow Sc \mid d$ This grammar is not immediately left-recursive, but it is still left-recursive.

$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}ca$ or

$\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac$ causes to a left-recursion

- So, we have to eliminate all left-recursions from our grammar

Algorithm

- *Input: Grammar G with no cycles or ε-productions*
- Arrange the nonterminals in some order $A1, A2, \ldots, An$

for $i = 1, \ldots, n$ **do**
    for $j = 1, \ldots, i\text{-}1$ **do**
        replace each

$$Ai \rightarrow Aj\ \gamma$$

        with

$$Ai \rightarrow \delta 1\ \gamma \mid \delta 2\ \gamma \mid \ldots \mid \delta k\ \gamma$$

        where

$$Aj \rightarrow \delta 1 \mid \delta 2 \mid \ldots \mid \delta k$$

    enddo
    eliminate the *immediate left recursion* in Ai
enddo

Example1

$S \to Aa \mid b$
$A \to Ac \mid Sd \mid f$

- Order of non-terminals: S, A

for S:

  - we do not enter the inner loop.
  - there is no immediate left recursion in S.

for A:

  - Replace $A \to Sd$ with $A \to Aad \mid bd$
    So, we will have $A \to Ac \mid Aad \mid bd \mid f$
  - Eliminate the immediate left-recursion in A

$$A \to bdA' \mid fA'$$
$$A' \to cA' \mid adA' \mid \varepsilon$$

So, the resulting equivalent grammar which is not left-recursive is:

$S \to Aa \mid b$
$A \to bdA' \mid fA'$
$A' \to cA' \mid adA' \mid \varepsilon$

Example2

$S \to Aa \mid b$
$A \to Ac \mid Sd \mid f$

- Order of non-terminals: A, S

for A:

- we do not enter the inner loop.
- Eliminate the immediate left-recursion in A

$A \to SdA' \mid fA'$
$A' \to cA' \mid \varepsilon$

for S:

- Replace $S \to Aa$ with $S \to SdA'a \mid fA'a$

So, we will have $S \to SdA'a \mid fA'a \mid b$

- Eliminate the immediate left-recursion in S

$S \to fA'aS' \mid bS'$
$S' \to dA'aS' \mid \varepsilon$

So, the resulting equivalent grammar which is not left-recursive is:

$S \to fA'aS' \mid bS'$
$S' \to dA'aS' \mid \varepsilon$
$A \to SdA' \mid fA'$
$A' \to cA' \mid \varepsilon$

Left-Factoring

- A predictive parser (a top-down parser without backtracking) insists that the grammar must be *left-factored*.

  grammar ➔ a new equivalent grammar suitable for predictive parsing

stmt → `if` expr `then` stmt `else` stmt |
`if` expr `then` stmt

- when we see `if`, we cannot now which production rule to choose to re-write *stmt* in the derivation.

Left-Factoring cont…

- In general,

  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ where α is non-empty and the first symbols of β_1 and β_2 (if they have one)are different.

- when processing α we cannot know whether expand

  A to $\alpha\beta_1$ or A to $\alpha\beta_2$

- But, if we re-write the grammar as follows

  $A \rightarrow \alpha A'$

  $A' \rightarrow \beta_1 \mid \beta_2$ so, we can immediately expand A to $\alpha A'$

Left-Factoring -- Algorithm

- For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix, let say

  $$A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$$

  convert it into

  $$A \rightarrow \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_m$$
  $$A' \rightarrow \beta_1 \mid \ldots \mid \beta_n$$

Left-Factoring – Example1

$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$

$\Downarrow$

$A \rightarrow aA' \mid \underline{cd}g \mid \underline{cd}eB \mid \underline{cd}fB$

$A' \rightarrow bB \mid B$

$\Downarrow$

$A \rightarrow aA' \mid cdA''$

$A' \rightarrow bB \mid B$

$A'' \rightarrow g \mid eB \mid fB$

Left-Factoring – Example2

$A \rightarrow ad \mid a \mid ab \mid abc \mid b$

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid b \mid bc$

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid bA''$

$A'' \rightarrow \varepsilon \mid c$

CFG - Terminology

- L(G) is *the language of G* (the language generated by G) which is a set of sentences.
- *A sentence of L(G)* is a string of terminal symbols of G.
- If S is the start symbol of G then
  - ω is a sentence of L(G) iff $S \Rightarrow \omega$ where ω is a string of terminals of G.
- The *language generated by G* is defined by

$$L(G) = \{w \in T^* \mid S \Rightarrow^+ w\}$$

CFG - Terminology

- If G is a context-free grammar, L(G) is a *context-free language*.
- Two grammars are *equivalent* if they produce the same language.
- $S \Rightarrow \alpha$ - If α contains non-terminals, it is called as a *sentential* form of G.
  - If α does not contain non-terminals, it is called as a *sentence* of G.

Non-Context Free Language Constructs

- There are some language constructions in the programming languages which are not context-free. This means that, we cannot write a context-free grammar for these constructions.

- L1 = { $\omega c \omega$ | ω is in $(a|b)^*$}   is not context-free

  ➔ declaring an identifier and checking whether it is declared or not later. We cannot do this with a context-free language. We need semantic analyzer (which is not context-free).

- L2 = $\{a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1\}$ is not context-free

  ➔ declaring two functions (one with n parameters, the other one with m parameters), and then calling them with actual parameters.