Grammars and Parsing
Article #14

Derivations (Text § 4.2)

We look at a description of a grammar as a way of **deriving** a language and how to design a grammar suitable for a recursive descent parser:

- our objective is a recursive descent parser where the parse tree is not actually generated but is implicit in the derivations
- a recursive descent parser can only work for certain types of grammars with the right kind of productions.

The derivation view is that the sentences of the language are generated by repeated application of the productions.

In general a string α can consist of a mixture of terminals and nonterminals.
A production that derives the string **α** from the nonterminal A has the form

$$A \rightarrow \alpha$$

Since strings are derived from other strings, a series of productions that generates $\boldsymbol{\alpha_n}$ from $\boldsymbol{\alpha_1}$ is written:

$$\alpha_1 => \alpha_2 => \alpha_3 => \ldots\ldots\ => \alpha_n$$

The symbol => means “generates in one step”
To summarize the above equation we can use the symbol =*> meaning “generates in 0 or more steps”. Thus

$$\alpha_1 =^{*}> \alpha_n \qquad \textbf{1.}$$

In general a string **α** is in **sentential** form if it can be generated by the grammar:

$$S =^{*}> \alpha$$

If a string **ω** contains only terminals, then it must have been derived from the start symbol by at least one production. We write this as:

$$S =^{+}> \omega$$

This equation defines the grammar. Such a string is a **sentence** in the language.

Derives in one step. The metasymbol for this is $\Rightarrow$.
We say a string v **derives** a string w if a production can be used to produce w from v. So

$$v \Rightarrow w$$

if we can write for some strings v and w (where x and z are other strings of terminals and nonterminals)

$$v = xYz, \quad w = xyz$$

where Y → y is a production in the grammar that produces the string y from the nonterminal Y.
If the strings x and v are empty, then we have

$$Y \Rightarrow y$$

Derives in one or more steps. The metasymbol for this is **=>+**

We say a string v derives a string w in one or more steps if w is produced from from v after applying one or many productions:

$$v \Rightarrow w_0 \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n = w$$

summarised as:

$$v \Rightarrow^+ w$$

Derives in zero or more steps The metasymbol for this is $\overset{*}{\Rightarrow}$
This is $\Rightarrow^+$ together with no derivation:

$v \overset{*}{\Rightarrow} w$ if $v \Rightarrow^+ w$ or $v = w$

Finally, we arrive at another way of thinking of a language.
Let $G[V_t, V_n, P, S]$ be a grammar (S is the start symbol).

- A string v is called a **sentential form** if v is derivable from S:

  $$S \overset{*}{\Rightarrow} v$$

- A string is a **sentence** if v consists only of terminals $v = V_t^+$
- The language $L(G[V_t,V_n,P,S])$ is therefore the set of sentences produced by the grammar:

  $$\mathbf{L(}G[V_t,V_n,P,S]\mathbf{) = \{v \mid S \overset{*}{\Rightarrow} v \text{ and } v \text{ in } T\}}$$

A language is the set of terminal strings (sentences) that is produced by the grammar. The sentences are a subset of all possible terminal strings.
]

leftmost and *rightmost* derivations

Consider the following grammar:

expr → expr op expr | (expr) | - expr | **id**
op → + | - | * | /

This first line is productions for **expr**. The second line is the productions for **op**. **expr** and **op** are nonterminals. **id** and **+, -, *** and / are terminals.

Consider this example from the grammar.
The string –(x + y) is a sentence of the grammar because:

leftmost derivation

expr => -expr => -(expr) => -(**expr** op expr) => -(**id** op expr)
=> -(x op expr) => -(x **+** expr) => -(x + **id**) => -(x + y)

or expr =*> -(x + y)

This is **leftmost** because the leftmost nonterminal is replaced at each stage.

The alternative is to replace the rightmost nonterminal at each stage:
rightmost derivation

expr => -expr => -(expr) => -(expr op **expr**) => -(expr op **id**)
=> -(expr op **y**) => -(expr **+** y) => -(**id** + y) => -(x + y)

leftmost and rightmost derivations generate the same parse tree and every parse tree has associated with it a unique leftmost and unique rightmost derivation.
(A type of parsing algorithm LL(1) that we will meet later takes its name from the fact that it reads the input token string from **L**eft (first L) to Right, does a **L**eftmost (second L) derivation, and looks one token ahead to determine the production to use)

Left and Right Associativity and Left and Right Recursion (Old Dragon Text § 2.2, New Dragon Text § 4.3.2)

Consider the following strings: 9 – 5 – 2 and a = b = c

1. 9 – 5 – 2

The – operator is **left associative** since **the 5 associates with the – on its left**. The normal interpretation of the expression is:

(9 – 5) – 2 = 2

not

9 – (5 – 2) = 6 wrong

2. a = b = c

The = operator is right associative since the b associates with the = on its right The normal interpretation of the expression is:

a = (b = c) c is assigned to b, then to a

not

(a = b) = c b may be undefined

How can this associativity be represented in grammars?

Consider two grammars that implement this associativity:

1. left associativity for strings of digits with arithmetic operators

list → **list** + digit | **list** – digit | digit
digit → [0-9]

Left associativity is enforced by evaluating left associating expressions lower down in the parse tree where precedence is higher.

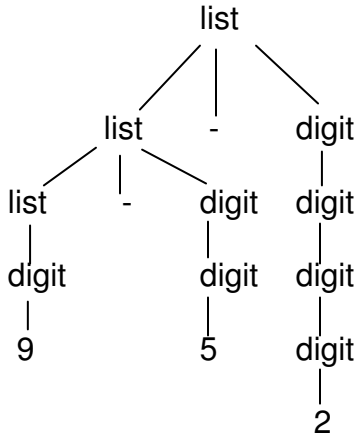
2. right associativity for strings of characters with assignment

right → letter = **right** | letter
letter → [a-z]

Right associativity is enforced by evaluating right associating expressions lower down in the parse tree where precedence is higher.

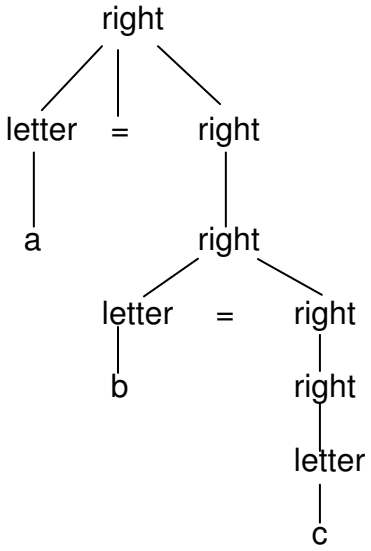
Parse trees generated for these expressions are (using a leftmost derivation and one derivation at each stage)):

1. 9 – 5 –2



Note how the left associative tree expands to the left

2. a = b = c



Note how the right associative tree grows down to the right.

Left and Right Recursion (Text § 2.4, 4.3)

Left-associative grammars naturally support the associativity of some operators, but there is a serious problem with recursive-descent parsers. Recursive descent parsers implement the productions as function calls. So if the nonterminal on the LHS is also the first symbol on the RHS, the function will enter infinite recursion.

In general productions look like:

left recursion **A → A α | β** **1.**

right recursion **A → α A | β** **2.**

where α and β are sequences of terminals and nonterminals that do not begin with A.
(In general a grammar is left-recursive if it has a nonterminal such that there is a derivation:

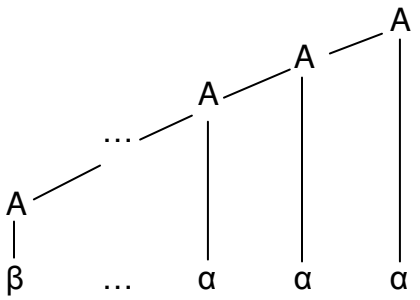
$A =^{+}> A \alpha$)

The strings generated by these productions are of the form:

1.

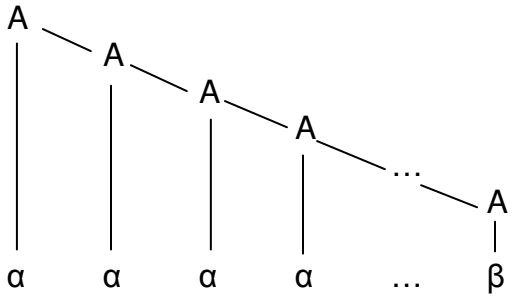
A => A α => A α α => A α α α => …….. β α α …α

Parse tree:



2.

A => α A => α α A => α α α A => …….. α α …α β



Note that for 1. a recursive-descent parser that attempts to apply the production as a function call would result in the evaluation of the first symbol on RHS which is the same function call, and the beginning of infinite recursion.

Hence in such a parser, we need to remove left recursion but implement the code so as to preserve left associativity.

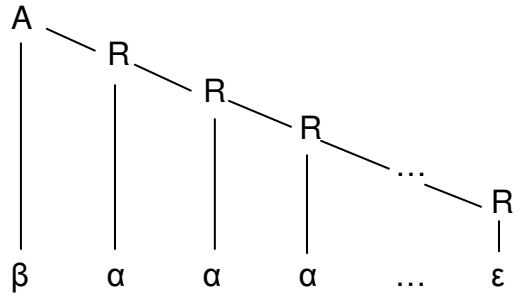
Eliminating Left Recursion **(Text § 2.4, 4.3)**

We can convert the left recursion (that follows naturally from the grammar) into a right recursion with the following algorithm that can be applied to any production and does not change the set of derivable strings.

original: $A \rightarrow A\ \alpha \mid \beta$

converted: $A \rightarrow \beta\ R$
$R \rightarrow \alpha\ R \mid \varepsilon$

Note that we now have right-recursion and an ε-production, but the strings generated are the same as the left recursive case:



Example.
Take the left-recursive part of the grammar above

list → **list** + digit | **list** – digit | digit

as a set of separate productions:

list → list + digit | digit
list → list – digit | digit

For a single alternationn

list → list + digit | digit

with left recursion removed

list → digit list'

list' → + digit list'| ε

For both alternations

list → list + digit | digit

list → list – digit | digit

with left recursion removed

list → digit list'

list' → + digit list' | – digit list' | ε

The general rule is:

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid .. \mid A\alpha_m \mid \beta_1 \mid \mid \beta_2 \mid .... \mid \beta_n$

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid ... \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid ... \mid \alpha_m A' \mid \varepsilon$

Note: these are examples of **immediate** left recursion.
The recursion could be several steps later:

S → A a | b
A → A c | S d | ε

Left Factoring a Grammar **(Text § 2.4, 4.3)**

The grammar

relational expression → primary expression > primary expression
| primary expression < primary expression

is not suitable for predictive parsing. It is not possible to make the decision which of the alternatives to chose having one token lookahead because all production alternatives have the same left-most terminal. It is possible to rework the grammar so that the decision can deferred until the parser can make the right choice. The corresponding grammar transformation is called ***left factoring***.

The general algorithm for left-factoring a grammar is

Given a grammar

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid ... \mid \alpha\beta_n \mid \gamma$

if α is not ε the left factored grammar is

$A \rightarrow \alpha A' \mid \gamma$
$A' \rightarrow \beta_1 \mid \beta_2 \mid ... \mid \beta_n$

This transformation must be applied repeatedly until no alternatives for a nonterminal have a common prefix.

In many cases the grammar may be transformed without applying the general rule. For example the grammar above can be rewritten as

relational expression →
primary expression relational operator primary expression

relational operator → < | >

Building the FIRST and FOLLOW sets for a Grammar (Text § 4.4)

The construction of a predictive parser requires two set of tokens to be built. The **FIRST** set is a set of tokens that appear at the left-most position after zero or more derivations are applied to a grammar production. The formal definition is

FIRST(A) = { a | $A \overset{*}{\Rightarrow} \mathbf{a}\alpha$ }

The following rules for computing the **FIRST** set can be derived from the formal definition.

BFTS1. If X is a terminal, then **FIRST(X) = {X}**

BFTS2. If X is a nonterminal and $X \rightarrow \varepsilon$ is a production, then add ε to **FIRST(X)** .

BFTS3. If X is a nonterminal and $X \rightarrow Y_1Y_2\ldots Y_k$ is a production then
FIRST(X) = {FIRST(Y_1)} if Y_1 does not derive **ε;**
FIRST(X) = {FIRST(Y_1), FIRST(Y_2)} if Y_1 derives **ε;**
FIRST(X) = {FIRST(Y_1), FIRST(Y_2), FIRST(Y_3)} if Y_1 and Y_2 derive **ε;**
FIRST(X) = {FIRST(Y_1), FIRST(Y_2), FIRST(Y_3), …FIRST(Y_i)} if **ε** is in all **FIRST(Y_1), FIRST(Y_2), FIRST(Y_3), …FIRST(Y_{i-1})**;
FIRST(X) = {FIRST(Y_1), FIRST(Y_2), FIRST(Y_3), …FIRST(Y_k), ε } if **ε** is in all **FIRST(Y_1), FIRST(Y_2), FIRST(Y_3), …FIRST(Y_k)**;

BFTS4. If X is a nonterminal and $X \rightarrow \alpha_1 \mid \alpha_2\ldots \mid \alpha_m$ is a production then
FIRST(X) = {FIRST(α_1), FIRST(α_1),…, FIRST(α_k) }.

The **FOLLOW** set is a set of tokens that appear on the right side of a nonterminal after zero or more derivations a grammar production. The formal definition is

FOLLOW(B) = { **a** | A $\overset{*}{\Rightarrow}$ α**B**a**a**γ }

The following rules for computing the **FOLLOW** set can be derived from the formal definition.

BFWS0. The **FOLLOW** set can not contain **ε**.
BFWS1. If **S** is the start symbol (nonterminal) for a grammar place **$** in the **FOLLOW(S),** where **$** is the input end-marker (for example, end of file)
BFWS2. If there is production A → α**B**a | α**B**b, then **FOLLOW(B) = {a,b}**
BFWS3. If there is production A → α**B**β and β does not derive **ε**, then add **FIRST(**β**)** to the **FOLLOW(B)** set.
BFWS4. If there is production A → α**B,** then add **FOLLOW(A)** to the **FOLLOW(B)** set
BFWS5. If there is production A → α**B**β and β derives **ε**, then add **FOLLOW(A)** to the **FOLLOW(B)** set.