### **Homework 5 – CSCI 330**

**Name:** Fathaabdi24  
**Professor:** Sarnath Ramnath  
**Assignment:** Homework 5  
**Due:** Monday, February 17th

**Q1. Derivations**

**String 1: ixoyowdssbes**

**Grammar**:

mathematica

I → iES | iESeS

E → EoG | G

G → x | y | z | w

S → s | dLb

L → Ls | sL

**Derivation**:

css

I → iESeS

E → EoG → EoGoG → GoGoG → xoy

→ i x o y o S e S

S → w

→ i x o y o w e S

S → dLb

L → sL → ss

→ d s s b

Final: ixoyowdssbes

#### **String 2: ixoys**

**Derivation**:

mathematica

I → iES

E → EoG → GoG → xoy

S → s

Final: ixoys

### **Q2. Eliminate Left Recursion**

We are given this grammar:

I → iES | iESeS

E → EoG | G

G → x | y | z | w

S → s | dLb

L → Ls | sL

### **Step 1: Identify Left Recursion**

Left recursion happens when a rule calls itself first, like:

E → EoG ← Left recursive

L → Ls ← Left recursive

L → sL ← Also left recursive (indirect loop)

### **Step 2: Remove Left Recursion**

#### **For E → EoG | G**

We rewrite it like this:

E → G E'

E' → oG E' | ε

Now E starts with G (not itself), and E' handles the repetition.

**For L → Ls | sL**

This is **both left and right recursive**, which makes it **not suitable for recursive descent parsing**.

We simplify L by allowing only one version — we can choose right-recursive:

L → sL | s

Or left-recursive:

L → Ls | s

But for **recursive descent**, we need **no left recursion**, so we use:

L → s L'

L' → L | ε

(Or simplify even further as L → s L', where L' → s L' | ε)

### **Final (Non-left-recursive) Grammar:**

I → i E S | i E S e S

E → G E'

E' → o G E' | ε

G → x | y | z | w

S → s | d L b

L → s L'

L' → s L' | ε

### **Q3. Recursive Descent Parsing Functions**

Each **non-terminal** in the grammar becomes a **function** in our parser. The parser looks at the **next symbol** in the input string and decides which rule to apply.

We assume the grammar is already **non-left-recursive and left-factored** (from Q2).

### **Function: parse-I**

Grammar rule:

I → i E S | i E S e S

**Logic**:

* If the next symbol is i, consume it
* Call parse-E, then parse-S
* If the next symbol is e, consume it and call parse-S again

### **Function: parse-E**

Grammar:

E → G E'

**Logic**:

* Call parse-G
* Then call parse-E'

### **Function: parse-E'**

Grammar:

E' → o G E' | ε

**Logic**:

* If the next symbol is o, consume it
* Then call parse-G, then parse-E' again
* If it’s not o, return ε (do nothing)

### **Function: parse-G**

Grammar:

G → x | y | z | w

**Logic**:

* If the next symbol is x, y, z, or w, consume it
* Otherwise, it’s an error

**Function: parse-S**

Grammar:

S → s | d L b

**Logic**:

* If next symbol is s, consume it
* If next symbol is d, consume it, call parse-L, then consume b

**Function: parse-L**

Grammar:

L → s L'

**Logic**:

* If next symbol is s, consume it and call parse-L'
* Otherwise, error

### **Function: parse-L'**

Grammar:

L' → s L' | ε

**Logic**:

* While the next symbol is s, keep consuming it and calling parse-L'
* Otherwise, return ε (do nothing)

### **Does every function have a unique action?**

Yes — after fixing left recursion and factoring, each function knows exactly what to do based on the next input symbol. There is **no ambiguity**.

### 

### **Q4. Left Factoring**

### **What is Left Factoring?**

Left factoring is used when two or more rules share the same beginning.  
This makes it hard for a parser to decide which rule to pick based on the next symbol.

We rewrite the rules so that the shared prefix is factored out.

**Step 1: Check for Common Prefixes**

Original rule:

I → iES | iESeS

Both options **start with iES**This **fails the pairwise disjointness test** because the parser doesn’t know which to pick when it sees it.

### **Step 2: Left Factor the Rule**

We rewrite **I** to factor out the common part:

I → i E S I'

I' → e S | ε

This way, we only look ahead **one symbol** at a time, and I' handle the difference.

### 

### **Final Grammar After Left Factoring**

I → i E S I'

I' → e S | ε

E → G E'

E' → o G E' | ε

G → x | y | z | w

S → s | d L b

L → s L'

L' → s L' | ε

**Q5. Updated Parser Functions (After Left Factoring)**

Now that we fixed **left recursion** and **common prefixes**, every parser function can choose a rule just by looking at **one symbol**.

Here’s how each function behaves:

### **parse-I**

Rule:

I → i E S I'

**Logic**:

* If symbol is i, consume it
* Then call parse-E, parse-S, and parse-I'

**parse-I'**

Rule:

I' → e S | ε

**Logic**:

* If symbol is e, consume it and call parse-S
* Otherwise, return ε (do nothing)

### **parse-E**

Rule:

E → G E'

**Logic**:

* Call parse-G then parse-E'

### **parse-E'**

Rule:

E' → o G E' | ε

**Logic**:

* If symbol is o, consume it and call parse-G, then parse-E'
* Else, return ε

### **parse-G**

Rule:

G → x | y | z | w

**Logic**:

* If symbol is one of x, y, z, or w, consume it
* Otherwise, it's an error

### **parse-S**

Rule:

S → s | d L b

**Logic**:

* If symbol is s, consume it
* If symbol is d, consume it, call parse-L, then consume b
* Else, error

### **parse-L**

Rule:

L → s L'

**Logic**:

* If symbol is s, consume it and call parse-L'
* Else, error

### **parse-L'**

Rule:

L' → s L' | ε

**Logic**:

* While next symbol is s, consume and call parse-L'
* If not s, return ε