3) Removing Empty Strings

Ensure only top-level symbol can be nullable

program ::= stmtSeq
stmtSeq ::= stmt | stmt ; stmtSeq
stmt ::= “” | assignment | whileStmt | blockStmt
blockStmt ::= { stmtSeq }
assignment ::= expr = expr
whileStmt ::= while (expr) stmt
expr ::= identifier

How to do it in this example?

3) Removing Empty Strings - Result

program ::= “” | stmtSeq

stmtSeq ::= stmt | stmt ; stmtSeq |

| ; stmtSeq | stmt ; | ;

stmt ::= assignment | whileStmt | blockStmt

blockStmt ::= { stmtSeq } | { }

assignment ::= expr = expr

whileStmt ::= while (expr) stmt

whileStmt ::= while (expr)

expr ::= identifier

3) Removing Empty Strings - Algorithm

- Compute the set of nullable non-terminals
- for each rule X ::=$s_1 \cdots s_n$ (n > 1) and each nullable s_i
  - add X::= $s_1 \cdots s_{i-1} s_{i+1} \cdots s_n$ if it doesn't exist
- Repeat the above step for the newly added rules
- Remove all empty right-hand sides
- If starting symbol S was nullable, then introduce a new start symbol S' instead, and add rule S' ::= S | ""

Note the no. of right-hand sides of X could be $O(2^n)$

3) Removing Empty Strings

- Since stmtSeq is nullable, the rule
  blockStmt ::= { stmtSeq }
  gives
  blockStmt ::= { stmtSeq } | { }
- Since stmtSeq and stmt are nullable, the rule
  stmtSeq ::= stmt | stmt ; stmtSeq
  gives
  stmtSeq ::= stmt | stmt ; stmtSeq
  | ; stmtSeq | stmt ; | ;

4) Eliminating unit productions

- Single production is of the form

  X ::=Y

where X,Y are non-terminals

```
program ::= stmtSeq
stmtSeq ::= stmt
            | stmt ; stmtSeq
stmt ::= assignment | whileStmt
assignment ::= expr = expr
whileStmt ::= while (expr) stmt
```

4) Unit Production Elimination Algorithm

- If there is a unit production

  X ::=Y put an edge (X,Y) into graph

- If there is a path from X to Z in the graph, and there is rule Z ::= s_1 s_2 … s_n then add rule

  X ::= s_1 s_2 … s_n

At the end, remove all unit productions.

4) Eliminate unit productions - Result

program ::= expr = expr | while (expr) stmt
| stmt ; stmtSeq

stmtSeq ::= expr = expr | while (expr) stmt
| stmt ; stmtSeq

stmt ::= expr = expr | while (expr) stmt

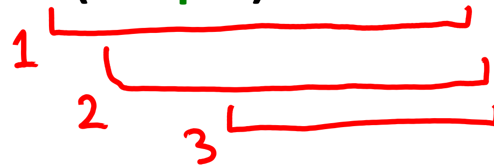
assignment ::= expr = expr

whileStmt ::= while (expr) stmt

now unreachable

5) Reducing Arity: No more than 2 symbols on RHS

stmt ::= while (expr) stmt



becomes

$stmt$ ::= while $stmt_1$

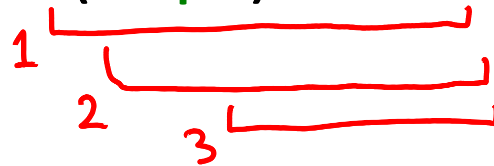
$stmt_1$ ::= ($stmt_2$

$stmt_2$ ::= expr $stmt_3$

$stmt_3$ ::=) stmt

6) A non-terminal for each terminal

stmt ::= while (expr) stmt



becomes

$stmt ::= N_{while}\ stmt_1$

$stmt_1 ::= N_(\ stmt_2$

$stmt_2 ::= expr\ stmt_3$

$stmt_3 ::= N_)\ stmt$

$N_{while} ::= while$

$N_(::= ($

$N_) ::=)$

Order of steps in conversion to CNF

1. remove unproductive symbols (optional)
2. remove unreachable symbols (optional)
3. make terminals occur alone on right-hand side
4. Reduce arity of every production to <= 2
5. remove epsilons
6. remove unit productions X::=Y
7. unproductive symbols
8. unreachable symbols

– What if we swap the steps 4 and 5 ?
  - Potentially exponential blow-up in the # of productions

Ordering of Unreachable / Unproductive symbols

First Unreachable then Unproductive

S := B C | “”
C := D
D := a
R := r

S := B C | “”
C := D
D := a

S := “”
C := D
D := a

First Unproductive then Unreachable

S := B C | “”
C := D
D := C
R := r

S := “”
C := D
D := a
R := r

S := “”

Exercises on LL(1) grammars

Exercise 1:

Consider a grammar for expressions where the multiplication sign is optional.

ex ::= ex + ex | ex * ex | ex ex |ID

- Find an LL(1) grammar recognizing the same language
- Create the LL(1) parsing table.

Exercise 1 – Solution

- First let's make the grammar unambiguous by associating precedence with operators
- In the process we also made sure that the grammar does not have left recursion
- ex ::= S + ex | S
- S ::= ID * S | ID S | ID
- Left factorization:
- ex ::= S Z
- Z ::= + ex | ϵ
- S ::= ID Z2
- Z2 ::= * S | S | ϵ

Exercise 1 – LL(1) parsing table

- ex ::= S Z EOF
- Z ::= + ex | ϵ
- S ::= ID Z2
- Z2 ::= * S | S | ϵ
- First let's compute first and follow sets after adding EOF to the end of the start symbol productions
  - First(ex) = First(S) = { ID }
  - First(Z) = { + }   First(Z2) = { * , ID }
  - Follow(ex) = Follow(Z) = { EOF }
  - Follow(S) = Follow(Z2) = { EOF, + }
- Z and Z2 are nullable

LL(1) parsing table

1. ex ::= S Z
2. Z ::= + ex
3. Z ::= ϵ
4. S ::= ID Z2
5. Z2 ::= * S
6. Z2 ::= S
7. Z2 ::= ϵ

	ID	+	*	EOF
ex	1	Error	Error	Error
Z	Error	2	Error	3
S	4	Error	Error	Error
Z2	6	7	5	7

Exercise 2

Prove that every LL(1) grammar is unambiguous.

Intuition:

There is a unique way to derive / parse a string because, for any given non-terminal, at most one alternative is applicable on scanning an input character.

This means we have unique left most derivations / parse trees for every string

Formal proof is presented in the next slide

Solution to Exercise 2 [Cont.]

Claim : Every string w derivable from every non-terminal N has a unique left most derivation.

- Proof by contradiction: Let D_1: $N \Rightarrow^* w$ and D_2: $N \Rightarrow^* w$ be two derivations for w
- D_1 and D_2 should diverge at some point. That is there exists a step at which a non-terminal expanded to different alternatives in the derivations.
- Let x we be prefix of w that is derived just before the point where D_1 and D_2 diverge. That is
  - $D_1 : N \Rightarrow^* xA\alpha \Rightarrow x\beta\alpha \Rightarrow^* w$
  - $D_2 : N \Rightarrow^* xA\alpha \Rightarrow x\gamma\alpha \Rightarrow^* w$,
- where A is a non-terminal, and α, β, γ are sequence of terminals and non-terminals, and $\beta \neq \gamma$
- If $x = w$ then $\beta\alpha \Rightarrow^* \epsilon$ and $\gamma\alpha \Rightarrow^* \epsilon$. Hence, there are two nullable alternatives for A which is a contradiction

Solution to Exercise 2 [Cont.]

- Therefore, say $|x| < |w|$. This implies that the next input character is $w_{|x|+1} = a\ (say)$
- Informally this means that both $A \rightarrow \gamma$ and $A \rightarrow \beta$ are applicable on seeing the input character a which contradicts the LL(1) property.
- Formally, given $a \in first(\beta\alpha)$ and $a \in first(\gamma\alpha)$

1. If both β and γ reduce to empty string (ϵ) in the derivations D_1 and D_2 then there are two nullable productions for A, which is a contradiction
2. If one of β and γ reduce to empty string and other doesn't
   - Let $\beta \Rightarrow^* \epsilon$ and γ derive a non-empty string
   - Since $a \in first(\gamma\alpha)$ and γ derives non-empty string, $a \in first(\gamma)$, which also implies that $a \in first(A)$
   - Since $a \in first(\beta\alpha)$ and β derives empty string, $a \in first(\alpha)$
   - Since $N \Rightarrow^* xA\alpha$, $first(\alpha) \subseteq follow(A)$. Hence, $a \in follow(A)$
   - Thus, $a \in follow(A) \cap first(A)$ and A is nullable, which contradicts LL(1) property
3. Finally, if both β and γ derive non-empty strings then $a \in first(\beta) \cap first(\gamma)$ again contradicting LL(1) property

Corollary of the proof

- The preceding proof not just proves that every string has a unique left most derivation in a LL(1) grammar but also proves the following:
- If two strings u and v share a common prefix 'x' , then the derivations of u and v cannot diverge before generating the prefix 'x'.
- That is the derivations of u and v should be of the form:
  - $S \Rightarrow^{*} \; x\,\alpha \Rightarrow^{*} \; u$
  - $S \Rightarrow^{*} \; x\,\alpha \Rightarrow^{*} \; v$

Exercise 3

Say that a grammar has a cycle if there is a *reachable, productive* non-terminal A such that $A \Rightarrow^{+} A$, i.e. it is possible to derive the nonterminal A from A by a nonempty sequence of production rules.

Show that if a grammar has a cycle, then it is not LL(1).

Solution to Exercise 3

- We proved before that LL(1) grammars are not ambiguous
- Consider a left most derivation D that contains A
- D: $S \Rightarrow^* \; xA\beta \Rightarrow^* \; w$
  - Where, x is a (possibly empty) sequence of terminals and
  - β is a sentential form
  - Such a derivation must exist as A is reachable (from the start symbol) and also productive
- Using A $\Rightarrow^+$A, we can derive another derivation for w
- D': $S \Rightarrow^* \; xA\beta \Rightarrow^+ xA\beta \Rightarrow^* \; w$
- There exists two left most derivations and hence two parse trees for w
- The grammar is ambiguous and hence cannot be LL(1)

Exercise 4

Show that the regular languages can be recognized with LL(1) parsers. Describe a process that, given a regular expression, constructs an LL(1) parser for it.

Solution for Exercise 4

- Let the DFA for the regular language be A : $(\Sigma, Q, q_0 \,, \delta, F)$
- Define a grammar G: (N, T, P, S) where,
- N = $\{ S_i \mid 1 \leq i \leq |Q| \}$
- $\mathrm{T} = \Sigma$
- S = S_0
- $\delta(q_i, a) = q_j \Rightarrow S_i \rightarrow a\, S_j \in P$
- $q_i \in F \Rightarrow S_i \rightarrow \epsilon \in P$

L(A) = L(G)

Exercise 5

Show that the language $\{ a^n b^m \mid n > m\}$ cannot have an LL(1) grammar ?

Note that the following grammar recognizes the language but is not LL(1)

S -> a S | P

P -> a P b | a

This question interesting but is quite difficult. A proof for this is provided in a separate pdf file in the lara wiki.

This is meant only as a supplementary material to provide more insights into LL(1) grammars.

It is not essential to fully understand the proof of this question.