Associativity and Precedence

Dealing with Associativity and Precedence

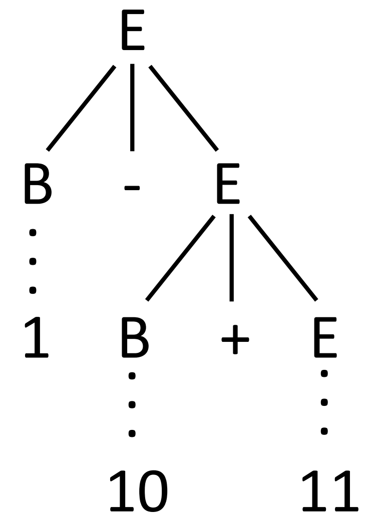
- CFGs generate *balanced parentheses and implicit order of evaluating expressions*
  - in the absence of parentheses
- *associativity*: grouping equivalent operations
  - example: 6 - 3 + 4
  - is it read as (6 - 3) + 4 or 6 - (3 + 4)?
  - we want left associativity, i.e. evaluate from left to right
- *precedence*: grouping non-equivalent symbols
  - example: 6 + 3 * 4
  - is it read as (6 + 3) * 4 or 6 + (3 * 4)?
  - we want multiplication to have precedence over addition

Associativity

Associativity of Expressions

- Recall this grammar.

1. $E \rightarrow B + E$
2. $E \rightarrow B - E$
3. $E \rightarrow B$
4. $B \rightarrow 0$
5. $B \rightarrow D$
6. $D \rightarrow 1$
7. $D \rightarrow D0$
8. $D \rightarrow D1$



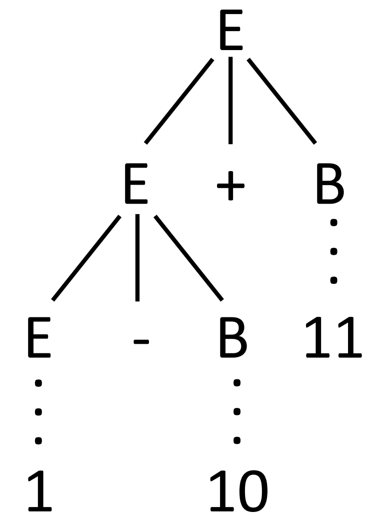
- Consider the tree corresponding to $E \Rightarrow B - E \Rightarrow B - B + E$
- The expression gets longer by adding more operators and digits (i.e. expressions) on the *right* hand side.
- Since the children get evaluated before the parent, 10 + 11 will be evaluated before 1 - ( )
- These rule enforce associativity from the *right*, i.e. 1 - (10+11)

Associativity

Associativity of Expressions

- Swap order of E and B in 1, 2.

1. $E \rightarrow E + B$
2. $E \rightarrow E - B$
3. $E \rightarrow B$
4. $B \rightarrow 0$
5. $B \rightarrow D$
6. $D \rightarrow 1$
7. $D \rightarrow D0$
8. $D \rightarrow D1$



- Consider the tree corresponding to $E \Rightarrow E + B \Rightarrow E - B + B$
- The expression gets longer by adding more operators and digits (i.e. expressions) on the *left* hand side.
- Since the children get evaluated before the parent, 1 - 10 will be evaluated before ( ) + 11
- These rule enforce associativity from the *left*, i.e. (1-10) +11

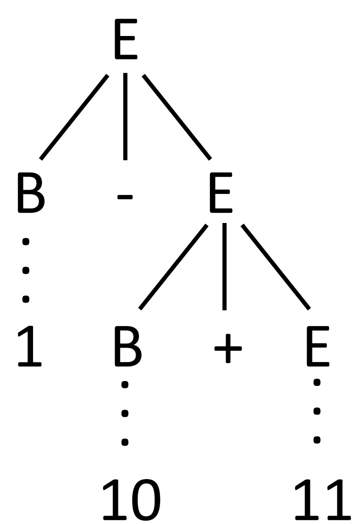
Associativity

When our grammar is *right recursive,* i.e.

1. $E \rightarrow B + E$
2. $E \rightarrow B - E$

our grammar becomes *right associative*, i.e.

$E \Rightarrow B - E \Rightarrow B - (B + E)$

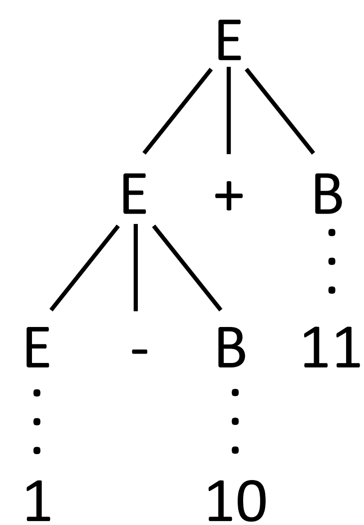


When our grammar is *left recursive,* i.e.

1. $E \rightarrow E + B$
2. $E \rightarrow E - B$

our grammar becomes *left associative,* i.e.

$E \Rightarrow E + B \Rightarrow (E - B) + B$

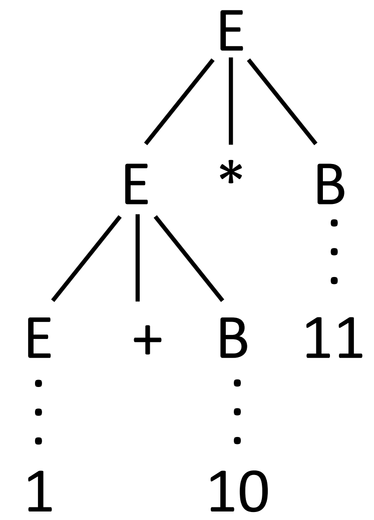


Precedence

Binary Expressions

- Now add multiplication and division.

1.	$E \rightarrow E + B$	6.	$B \rightarrow 0$
2.	$E \rightarrow E - B$	7.	$B \rightarrow D$
3.	$E \rightarrow E * B$	8.	$D \rightarrow 1$
4.	$E \rightarrow E / B$	9.	$D \rightarrow D0$
5.	$E \rightarrow B$	10.	$D \rightarrow D1$



- Consider the derivation $E \Rightarrow E * B \Rightarrow E + B * B$
- This grammar will evaluate the expression 1+10*11 as (1+10)*11 which *ignores the standard rules of precedence.*

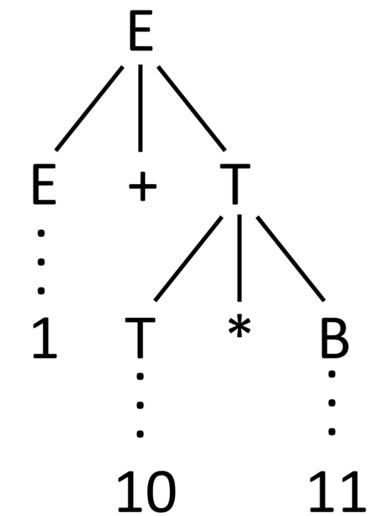
Precedence

Binary Expressions

- Introduce a new non-terminal T

1. $E \rightarrow E + T$
2. $E \rightarrow E - T$
3. $T \rightarrow T * B$
4. $T \rightarrow T / B$
5. $E \rightarrow T$
6. $T \rightarrow B$

6. $B \rightarrow 0$
7. $B \rightarrow D$
8. $D \rightarrow 1$
9. $D \rightarrow D0$
10. $D \rightarrow D1$



- Consider the derivation $E \Rightarrow E + T \Rightarrow E + T * B$
- This grammar will evaluate the expression 1 + 10 * 11 as 1+(10*11)
- The T nonterminal is farther from the start symbol E and will be evaluated before expressions involving E.

Topic 11 – Top-Down Parsing

Key Ideas

- Parsing
- Top-down and bottom-up parsing
- First(), Follow(), Empty()
- LL(1) Parsing

References

- *Basics of Compiler Design* by Torben Ægidius Mogensen sections 3.7 to 3.10, 3.12

Review

Review

- *Regular Languages*
  - e.g. the set of WLP4 tokens
  - e.g. the set of MIPS tokens
  - a*b, regular expressions, DFA, NFA, ε-NFA
  - relatively easy to scan
- *Context-free languages* add
  - check balanced parentheses, i.e. when does function end, for loop end, etc.
  - capture associativity, precedence
- How are the two related?

Regular Expressions and CFGs

All Regular Languages are Context-free

- *Basic Idea:* use a CFG to generate the components of the recursive definition of a Regular Expressions

Base Cases

- use $S \to \varepsilon$ to generate language $\mathcal{L} = \{ \varepsilon \}$
- use $S \to a$ to generate language $\mathcal{L} = \{ a \}$

Building up Expressions

- if E_1 and E_2 are regular expressions, with $S_1 \to E_1$ and $S_2 \to E_2$
  - $S \to S_1 S_2$ generates $E_1 E_2$ (*concatenation*)
  - $S \to S_1$; $S \to S_2$ generates $E_1 | E_2$ (*union*)
  - $S \to SS_1$; $S \to \varepsilon$ generates E_1^* (*repetition*)

Regular Expressions and CFGs

Exercise

- Create a CFG that derives the language a*b | (cd) *

Parsing

What is Parsing

- *Parsing:* Given a grammar *G* and a word *w*, *find a derivation* for *w.*
- *Our goal:* look at the characters in *w* and decide which rules derived *w* from the start symbol
- Two strategies
  1. *Top-down:* Find a non-terminal (e.g. X) and replace it with the right-hand side (e.g. aXb if X $\rightarrow$ aXb is a rule).
  2. *Bottom-up:* replace a right-hand side (e.g. aaXbb) with a non-terminal: (e.g. aXb if X $\rightarrow$ aXb is a rule).
- In both of the above strategies, we have to make the correct decision at each step.

Parsing Algorithms

Backtracking

- There is a *backtracking algorithm* for parsing in a CFG, i.e. try each rule in turn…

    if we can move "forward"

        do so

    else

        go back a step and try the next rule

    stop when we find a derivation

- Backtracking is not practical.
    - an exhaustive search
    - takes exponential time
    - will always find derivation, if it exists
- We will look at *two linear-time algorithms.*

Stack-based Parsing

Using a Stack

- For top-down parsing, we use a stack to remember information about our derivations and/or processed input.
- Recall that CFGs are recognized by a DFA with a stack
- e.g. for language of paired parentheses
  - if input is ‘(‘, push it on the stack
  - if input is ‘)’ pop the stack
  - if you *pop when the stack is empty*: ERROR
  - if the *stack is not empty when you are finished* processing the input: ERROR
- e.g (( ) ( ))
- because of the possibility of errors we need to *augment* our grammar …

Augmenting Grammars

Using a Stack

- We want to be able to detect both these situations easily
- We augment our grammars by adding
  - a character to *mark the beginning:* ⊢ (also called *BOF*)
  - a character to *mark the end:* ⊣ (also called *EOF*)
  - a *new start symbol* S' that only appears once

1. $S' \rightarrow \vdash S \dashv$
2. $S \rightarrow AyB$
3. $A \rightarrow ab$
4. $A \rightarrow cd$
5. $B \rightarrow z$
6. $B \rightarrow wz$

Exercise

Derive the word ⊢ abywz ⊣

S'	$\Rightarrow \vdash S \dashv$	rule (1)
	$\Rightarrow \vdash AyB \dashv$	rule (2)
	$\Rightarrow \vdash abyB \dashv$	rule (3)
	$\Rightarrow \vdash abywz \dashv$	rule (6)

Top-Down Parsing

Parsing Algorithm

- to start, push the start symbol, S’, on the stack
- when it is a *non-terminal* at the top of the stack:
  - *expand* the non-terminal using a production rule where the RHS of the rule matches the input
- when it is a *terminal* at the top: *match* with input
  - pop the terminal off of the stack
  - read the next character from the input

Top-Down Parsing

Parsing the Input

- To start, push S' on the stack

	Derivation	Read	Input	Stack	Action
1	S'		⊢ abywz ⊣	> S'	

- When it is a *non-terminal* at the top of stack: *expand* the non-terminal (using a production rule) so that the top of the stack matches the first symbol of the input.
  - in this case use rule 1 (S' $\rightarrow$ ⊢ S ⊣) because the first symbol of the input and the RHS of rule 1 is '⊢'

	Derivation	Read	Input	Stack	Action
1	S'		⊢ abywz ⊣	> S'	expand (1)
2	⊢ S ⊣		**⊢** abywz ⊣	> **⊢** S ⊣	

Top-Down Parsing

Parsing the Input

- Since the top of the stack matches the first char of the input, pop ⊢ off the stack and read the next char of the input

	Derivation	Read	Input	Stack	Action
2	⊢ S ⊣		⊢ abywz ⊣	> ⊢ S ⊣	match
3	⊢ S ⊣	⊢	abywz ⊣	> **S** ⊣	

- The top of the stack in a non-terminal so expand it using rule 2 ($S \rightarrow AyB$). There is only one choice of rule to use.

	Derivation	Read	Input	Stack	Action
3	⊢ S ⊣	⊢	abywz ⊣	> **S** ⊣	expand (2)
4	⊢ AyB ⊣	⊢	abywz ⊣	> **A** y B ⊣	

Top-Down Parsing

Parsing the Input

- The top of the stack in a non-terminal so expand it.
- There are two possible rules to use: 3 (A → ab) and 4 (A → cd) but only the RHS of rule 3 matches the input a.

	Derivation	Read	Input	Stack	Action
4	⊢ AyB ⊣	⊢	abywz ⊣	> **A** y B ⊣	expand (3)
5	⊢ AyB ⊣	⊢	**a**bywz ⊣	> **a** b y B ⊣	match

- Read from input and pop the next three chars, which match.

	Derivation	Read	Input	Stack	Action
6	⊢ abyB ⊣	⊢ a	**b**ywz ⊣	> **b** y B ⊣	match
7	⊢ abyB ⊣	⊢ ab	**y**wz ⊣	> **y** B ⊣	match
8	⊢ abyB ⊣	⊢ aby	wz ⊣	> **B** ⊣	

Top-Down Parsing

Parsing the Input

- Again, the top of the stack in a non-terminal so expand it.
- There are two possibilities: 5 B $\rightarrow$ z or 6 B $\rightarrow$ wz, but only the RHS of rule 6 matches the input w.

	Derivation	Read	Input	Stack	Action
8	⊢ abyB ⊣	⊢ aby	wz ⊣	> **B** ⊣	expand (6)
9	⊢ abywz ⊣	⊢ aby	wz ⊣	> **w** z ⊣	

- Pop off the stack and read the next two chars, which match.

	Derivation	Read	Input	Stack	Action
9	⊢ abywz ⊣	⊢ aby	wz ⊣	> **w** z ⊣	match
10	⊢ abywz ⊣	⊢ abyw	z ⊣	> **z** ⊣	match
11	⊢ abywz ⊣	⊢ abywz	⊣	> **⊣**	

Top-Down Parsing

Parsing the Input

- The last character in the input matches the last character on the stack, pop it off the stack and accept the string.

	Derivation	Read	Input	Stack	Action
11	⊢ abywz	⊢ abywz	⊣	> ⊣	match
12	⊢ abywz ⊣	⊢ abywz⊣		>	ACCEPT

- The next slide shows the complete parsing of abywz using the grammar:

1. $S' \rightarrow$ ⊢ S ⊣
2. $S \rightarrow AyB$
3. $A \rightarrow ab$
4. $A \rightarrow cd$
5. $B \rightarrow z$
6. $B \rightarrow wz$

Top-Down Parsing

Parsing the Input

	Derivation	Read	Input	Stack	Action
1	S'		⊢ abywz ⊣	> S'	expand (1)
2	⊢ S ⊣		**⊢** abywz ⊣	> **⊢** S ⊣	match
3	⊢ S ⊣	⊢	abywz ⊣	> **S** ⊣	expand (2)
4	⊢ AyB ⊣	⊢	abywz ⊣	> **A** y B ⊣	expand (3)
5	⊢ AyB ⊣	⊢	**a**bywz ⊣	> **a** b y B ⊣	match
6	⊢ abyB ⊣	⊢ a	**b**ywz ⊣	> **b** y B ⊣	match
7	⊢ abyB ⊣	⊢ ab	**y**wz ⊣	> **y** B ⊣	match
8	⊢ abyB ⊣	⊢ aby	wz ⊣	> **B** ⊣	expand (6)
9	⊢ abywz ⊣	⊢ aby	**w**z ⊣	> **w** z ⊣	match
10	⊢ abywz ⊣	⊢ abyw	**z** ⊣	> **z** ⊣	match
11	⊢ abywz ⊣	⊢ abywz	**⊣**	> **⊣**	ACCEPT

Top-Down Parsing

Top-down parsing with a stack

- *invariant* (i.e. true throughout entire process)

  derivation = input already read + stack (read from the top-down) , e.g.

  - Line 3: ⊢ S ⊣
  - Line 6: ⊢ a byB ⊣
  - Line 9: ⊢ aby wz ⊣
- How do we know when we are done?
  - both stack and input contain ⊣
- How do we know which rule to use?

  Our Goal: to be able to correctly predict which rule applies!

LL(1) Parsing

Meaning of LL(1)

- first 'L' means process the input from **L**eft to right
- second 'L' means find a **L**eftmost derivation
- 1 means algorithm is allowed to look ahead 1 token

Goal: Unambiguous Prediction

- Find what rule applies if *A* (a non-terminal) is on the stack and *a* (a terminal) is the next symbol in the input to be read
- For LL(1) grammars
  - for all non-terminals *A* and all terminals *a:* $|\text{Predict}(A, a)| \leq 1$
  - i.e. given an *A* on the top of the stack and an *a* as the next input character at most one rule can apply.
- Implement Predict(A, a) as a table.

LL(1) Parsing

How to Construct a Predictor Table: First()

- *ask:* For each non-terminal {S', S, A, B} what are the possible terminals they can derive on the leftmost side.

1. S' → ⊢ S ⊣
2. S → AyB
3. A → ab
4. A → cd
5. B → z
6. B → wz

	a	b	c	d	y	w	z	⊢	⊣
S'								1	
S	2		2						
A	3		4						
B						6	5		

- How to read table: if S' is on the stack and the next input is ⊢, the entry at (S', ⊢) is 1, so apply rule 1.
- Empty cells are error states.

LL(1) Parsing

Filling up a Row

- *Question:* What is the *first rule I apply* that *will eventually* derive the first non-terminal (i.e. the one on the left)
- How to fill a row: start with that row's non-terminal and try all applicable rules, tracking which terminal symbol appears as the first character on the left.

Row 1: S'

1. $S' \rightarrow \vdash S \dashv$ (rule 1)

Row 2: S

2. $S \rightarrow AyB$ (rule 2)
3. $A \rightarrow ab$
4. $A \rightarrow cd$

Row 3: A

3. $A \rightarrow ab$ (rule 3)
4. $A \rightarrow cd$ (rule 4)

Row 4: B

5. $B \rightarrow z$ (rule 5)
6. $B \rightarrow wz$ (rule 6)

LL(1) Parsing

How to Construct a Predictor Table: Follow()

- To understand Follow(), we need to add a rule where a non-terminal disappears, e.g. B $\rightarrow \varepsilon$

1. S' $\rightarrow$ ⊢ S ⊣
2. S $\rightarrow$ AyB
3. A $\rightarrow$ ab
4. A $\rightarrow$ cd
5. B $\rightarrow$ z
6. B $\rightarrow$ wz
7. B $\rightarrow \varepsilon$

	a	b	c	d	y	w	z	⊢	⊣
S'								1	
S	2		2						
A	3		4						
B						6	5		7

- We can have: S' $\Rightarrow$ ⊢ S ⊣ $\Rightarrow$ ⊢ AyB ⊣ $\Rightarrow$ ⊢ abyB ⊣ $\Rightarrow$ ⊢ aby ⊣
  - i.e. ⊣ *can appear after B but there is no rule B $\rightarrow$ ⊣*
  - the symbol "⊣" comes from the production 1: S' $\rightarrow$ ⊢ S ⊣

LL(1) Parsing

How to Construct a Predictor Table: Follow()

- The terminal symbol "⊣" is in Follow(B)
- Why: *there is a derivation from the start symbol S' $\Rightarrow^*$ $\alpha B\dashv\gamma$*, where
  - α and γ are (possibly empty) sequences of terminals and non-terminals
  - there is a rule $B \rightarrow \varepsilon$
  - the terminal"⊣" *does not* come after B because there is a derivation $B \Rightarrow \dashv\gamma$,
  - *but it can* follow after B, because there is a derivation from S' that puts "⊣" after B and then has B disappear (i.e. $B \rightarrow \varepsilon$).

How to Construct a Predictor Table

Calculating Empty(α)

- Sometimes called nullable(α)
- *Asking* if α can disappear?
- Here $B_i \in (N \cup T)$, $\alpha \in (N \cup T)^*$
- **Empty**(α) = true if $\alpha \Rightarrow^* \varepsilon$
  - False if α has a terminal in it, only non-terminals can disappear.
  - True if there is a rule $\alpha \rightarrow \varepsilon$
  - For any rule of the form $\alpha \rightarrow B_1 B_2 \ldots B_n$
    Empty(α) is true if each of Empty(B_1), Empty(B_2), ..., Empty(B_n) is true.

How to Construct a Predictor Table

Calculating Follow(A)

- *Asking:* Starting from the start symbol, does the terminal b ever occur immediately following A.
- Here $b \in T$, $A \in N$, $B_i \in (N \cup T)$, $\alpha, \beta \in (N \cup T)^*$
- **Follow**(A) = $\{b \mid S' \Rightarrow^* \alpha A b \beta\}$

Initialize by setting Follow(A) = { } // the empty set

for each rule of the form $A \rightarrow B_1B_2...B_n$:

for each n:

if (B_i is a non-terminal)

$Follow(B_i) = Follow(B_i) \cup First(B_{i+1}B_{i+2}...B_n)$

if ($Empty(B_1B_2...B_{i-1})$)

$Follow(B_i) = Follow(B_i) \cup Follow(A)$

How to Construct a Predictor Table

Calculating Predict(A, *a*)

- *Asking:* If A is on the top of the stack and *a* is the next symbol in the input, which rule should be used to expand A?
- Here $\alpha, \beta \in (N \cup T)^*$ $a \in T$ $A \in N$
- **Predict**(A, a) = $\{ A \rightarrow \alpha \mid a \in \text{First}(\alpha) \} \cup$

  $\{ A \rightarrow \beta \mid a \in \text{Follow}(A) \text{ and Empty}(\beta) = \text{true} \}$

LL(1) Parsing

```
Input: w
push: ⊢ S ⊣
for each a ∈ w
    while (top of stack is some non-terminal A ) {
        pop A
        if (Predict(A, a) == (A → α) {      // 1. Try to predict rule
            push α on stack (in reverse)
        else
            reject                           //    no rule found, error
        }
        pop c                                // 2. Try to match symbol
        if (c ≠ a) reject                    //    no match found
}
accept w
```

Example of LL(1) Parsing

LL(1) Parsing

	Derivation	Read	Input	Stack	Action
1	S’		⊢ abywz ⊣	> S’	predict(S’, ⊢) = 1
2	⊢ S ⊣		⊢ abywz ⊣	> ⊢ S ⊣	match
3	⊢ S ⊣	⊢	abywz ⊣	> **S** ⊣	predict(S, a) = 2
4	⊢ AyB ⊣	⊢	abywz ⊣	> **A** y B ⊣	predict(A, a) = 3
5	⊢ AywB ⊣	⊢	**a**bywz ⊣	> **a** b y B ⊣	match
6	⊢ abywB ⊣	⊢ a	**b**ywz ⊣	> **b** y B ⊣	match
7	⊢ abywB ⊣	⊢ ab	**y**wz ⊣	> **y** B ⊣	match
8	⊢ abywB ⊣	⊢ aby	wz ⊣	> **B** ⊣	predict(B,w) = 6
9	⊢ abywz ⊣	⊢ aby	**w**z ⊣	> **w** z ⊣	match
10	⊢ abywz ⊣	⊢ abyw	**z** ⊣	> **z** ⊣	match
11	⊢ abywz ⊣	⊢ abywz	⊣	> ⊣	ACCEPT

Non-LL(1) Grammars

A Non-LL(1) Grammar

G: 1. $S \rightarrow a\ b$

2. $S \rightarrow a\ c\ b$

- |L(G)| = 2, i.e. L(G) = {ab, acb}
- Not in LL(1).
- The predict table is ambiguous.

	a	b	c
S	1,2		

- Must look ahead to the second symbol in order to tell which rule to use. The predict table must consider pairs of terminals.
- G is in LL(2).

	aa	ab	ac	ba	bb	bc	ca	cb	cc
S		1	2						

Non-LL(1) Grammars

Converting a Non-LL(1) Grammar

LL(2)

G: 1. $S \rightarrow a\ b$

2. $S \rightarrow a\ c\ b$

LL(1)

G': 1'. $T \rightarrow a\ X$

2'. $X \rightarrow b$

3'. $X \rightarrow cb$

	a	b	c
T	1		
X		2	3

- Rewrite overlapping productions (1 and 2) so that
  - one rule contains the common prefix (a) and
  - a new non-terminal (X) produces the different suffixes (b and cb).