Session 22

Top-down Parsing

Parsing

- From Latin: Parts of the speech or grammatical categories
- If G is a CFG over Σ and $x \in \Sigma^*$, parsing x is the process of finding a derivation in G for x or determining that there is non.
- Antecedent: simulation of a derivation of G by a PDA
  - Top-down leftmost derivation
  - Bottom-up rightmost derivation
- However, these two are non-deterministic and do not provide an algorithm directly!

Parsing algorithms

- Trivial algorithm: explore all deterministic paths in certain order (i.e. de first depth or backtrack) and see if one path leads to acceptance
  - Too costly: out of the question!
- Confront non-determinism directly
  - Use all available information in the input to select the better choice (maybe deterministically)
  - Profit from the form or the grammar: consider the top of the stack and the next input symbol to determine the next move in the simulated derivation
- Two approaches:
  - Top-down
  - Bottom-up

Top-down *Lookahead* Parser

- Two moves in original simulation:
  - Move 1: If the symbol on top of the stack is a variable in the left side of a production, replace the variable by the corresponding right side of the production
  - If the input symbol matches the top of the stack, consume the symbol and pop!
- Eliminate non-determinism by looking ahead one symbol in the input string
  - In move one: look at the input symbol and make an intelligent choice of production
  - Choose a production (move 1) only when there is no other choice!

A Top-down simulation

L = Language of balanced parenthesis

$S \to T\$$
$T \to [T]T$
$T \to \Lambda$

Input string

Finite state control

The logic of the machine:
- Push parenthesis in the way in
- Pop parenthesis in the way in

A Top-down simulation

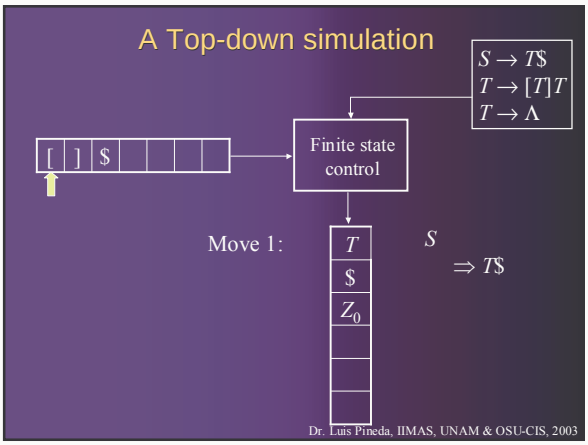
$S \to T\$$
$T \to [T]T$
$T \to \Lambda$

Finite state control

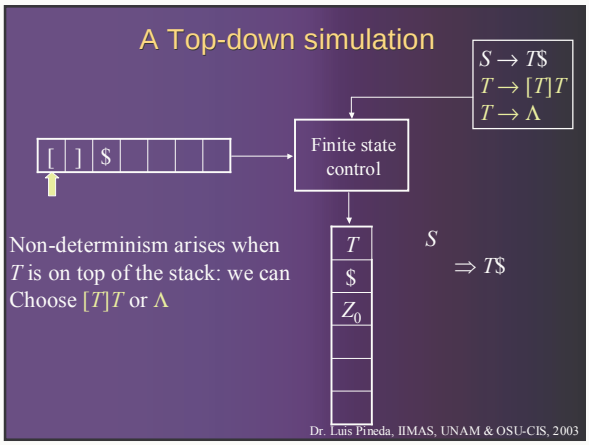
Move 0:

S

A Top-down simulation



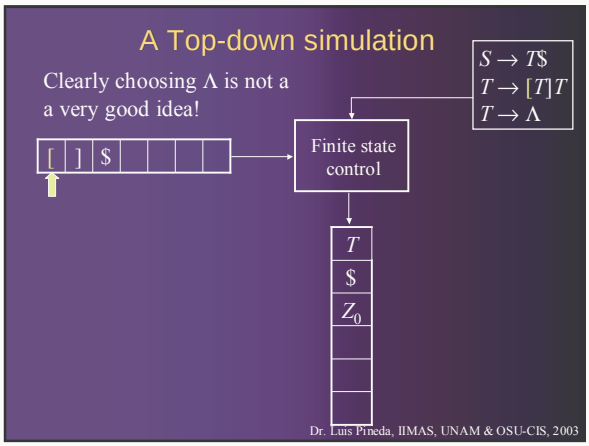
A Top-down simulation



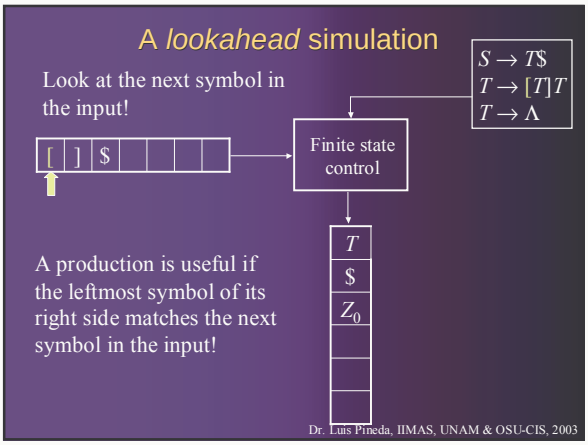
Nondeterministic PDA

Id	State	Input	Stack symbol	Move(s)
1	q_0	Λ	Z_0	(q_1, SZ_0)
2	q_1	Λ	S	(q_1, TZ_0)
3	q_1	Λ	T	$(q_1, [T]T)$, (q_1, Λ)
4	q_1	[	[	(q_1, Λ)
5	q_1	]	]	(q_1, Λ)
6	q_1	$	$	(q_1, Λ)
7	q_1	Λ	Z_0	(q_2, Z_0)
Other combinations				non

A Top-down simulation



A *lookahead* simulation



Lookahead

- Remove non-determinism by looking one symbol ahead in the input string
- Consider only the rules that have that symbol as its leftmost symbol in the right side of the corresponding production
- Design a deterministic PDA that corresponds to the non-deterministic one

Deterministic lookahead PDA

Id	State	Input	Stack symbol	Move(s)
1	q_0	Λ	Z_0	(q_1, SZ_0)
2	q_1	Λ	S	(q_1, TZ_0)
3	q_1	[	T	$(q_1, [T]T)$
4	q_1	Λ	[	(q_1, Λ)
5	q_1	]	T	(q_1, Λ)
6	q_1	Λ	]	(q_1, Λ)
7	q_1	$	T	(q_s, Λ)
8	q_s	Λ	$	(q_1, Λ)
9	q_1	[	[	(q_1, Λ)
10	q_1	]	]	(q_1, Λ)
11	q_1	$	$	(q_1, Λ)
12	q_1	Λ	Z_0	(q_2, Z_0)
Other combinations				non

Consume lookahead symbol

$T \to [T]T$

Id	State	Input	Stack symbol	Move(s)
1	q_0	Λ	Z_0	(q_1, SZ_0)
2	q_1	Λ	S	(q_1, TZ_0)
3	q_1	[	T	$(q_1, [T]T)$
4	q_1	Λ	[	(q_1, Λ)
5	q_1	]	T	(q_1, Λ)
6	q_1	Λ	]	(q_1, Λ)
7	q_1	$	T	(q_s, Λ)
8	q_s	Λ	$	(q_1, Λ)
9	q_1	[	[	(q_1, Λ)
10	q_1	]	]	(q_1, Λ)
11	q_1	$	$	(q_1, Λ)
12	q_1	Λ	Z_0	(q_2, Z_0)
Other combinations				non

Consume lookahead symbol

$T \to \Lambda$

Id	State	Input	Stack symbol	Move(s)
1	q_0	Λ	Z_0	(q_1, SZ_0)
2	q_1	Λ	S	(q_1, TZ_0)
3	q_1	[	T	$(q_1, [T]T)$
4	q_1	Λ	[	(q_1, Λ)
5	q_1	]	T	(q_1, Λ)
6	q_1	Λ	]	(q_1, Λ)
7	q_1	$	T	(q_s, Λ)
8	q_s	Λ	$	(q_1, Λ)
9	q_1	[	[	(q_1, Λ)
10	q_1	]	]	(q_1, Λ)
11	q_1	$	$	(q_1, Λ)
12	q_1	Λ	Z_0	(q_2, Z_0)
Other combinations				non

lookahead the $ symbol

$T \to \Lambda$

Id	State	Input	Stack symbol	Move(s)
1	q_0	Λ	Z_0	(q_1, SZ_0)
2	q_1	Λ	S	(q_1, TZ_0)
3	q_1	[	T	$(q_1, [T]T)$
4	q_1	Λ	[	(q_1, Λ)
5	q_1	]	T	(q_1, Λ)
6	q_1	Λ	]	(q_1, Λ)
7	q_1	$	T	(q_s, Λ)
8	q_s	Λ	$	(q_1, Λ)
9	q_1	[	[	(q_1, Λ)
10	q_1	]	]	(q_1, Λ)
11	q_1	$	$	(q_1, Λ)
12	q_1	Λ	Z_0	(q_2, Z_0)
Other combinations				non

Pop operations

Id	State	Input	Stack symbol	Move(s)
1	q_0	Λ	Z_0	(q_1, SZ_0)
2	q_1	Λ	S	(q_1, TZ_0)
3	q_1	[	T	$(q_1, [T]T)$
4	q_1	Λ	[	(q_1, Λ)
5	q_1	]	T	(q_1, Λ)
6	q_1	Λ	]	(q_1, Λ)
7	q_1	$	T	(q_s, Λ)
8	q_s	Λ	$	(q_1, Λ)
9	q_1	[	[	(q_1, Λ)
10	q_1	]	]	(q_1, Λ)
11	q_1	$	$	(q_1, Λ)
12	q_1	Λ	Z_0	(q_2, Z_0)
Other combinations				non

deterministic lookahead PDA

Id	State	Input	Stack symbol	Move(s)
3	q_1	[	T	$(q_1, [T]T)$
X	q_1	Λ	T	(q_1, Λ)

No rule entry in the table for: $T \to \Lambda$

A *lookahead* simulation

Look at the next symbol in the input!

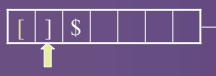
Finite state control

$S \to T\$$
$T \to [T]T$
$T \to \Lambda$

A production is useful if the leftmost symbol of its right side matches the next symbol in the input!

A *lookahead* simulation

Use the lookahead!



Finite state control

$S \to T\$$
$T \to [T]T$
$T \to \Lambda$

Consume the symbol and Push right-side of production!

S
$\Rightarrow T\$$
$\Rightarrow [T]T\$$

A *lookahead* simulation

Finite state control

$S \to T\$$
$T \to [T]T$
$T \to \Lambda$

Here, we have to remember that we need a "]" to balance the "[" that was consume in the *lookahead* move: the machine go to state $q_{[}$

S
$\Rightarrow T\$$
$\Rightarrow [T]T\$$

Consume lookahhead symbol

$T \to [T]T$

Id	State	Input	Stack symbol	Move(s)
1	q_0	Λ	Z_0	(q_1, SZ_0)
2	q_1	Λ	S	(q_1, TZ_0)
3	q_1	[	T	$(q_{[}, [T]T)$
4	$q_{[}$	Λ	[	(q_1, Λ)
5	q_1	]	T	$(q_{]}, \Lambda)$
6	$q_{]}$	Λ	]	(q_1, Λ)
7	q_1	$	T	$(q_{\$}, \Lambda)$
8	$q_{\$}$	Λ	$	(q_1, Λ)
9	q_1	[	[	(q_1, Λ)
10	q_1	]	]	(q_1, Λ)
11	q_1	$	$	(q_1, Λ)
12	q_1	Λ	Z_0	(q_2, Z_0)
Other combinations				non

A *lookahead* simulation

Finite state control

$S \to T\$$
$T \to [T]T$
$T \to \Lambda$

Pop:

S
$\Rightarrow T\$$
$\Rightarrow [T]T\$$

A *lookahead* simulation

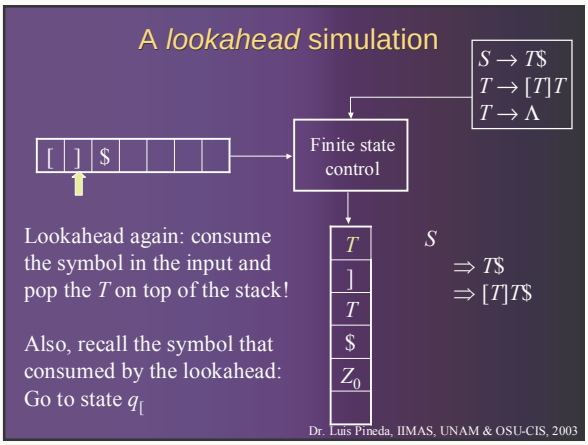
Finite state control

$S \to T\$$
$T \to [T]T$
$T \to \Lambda$

After pop:

S
$\Rightarrow T\$$
$\Rightarrow [T]T\$$

A *lookahead* simulation



Lookahead again: consume the symbol in the input and pop the T on top of the stack!

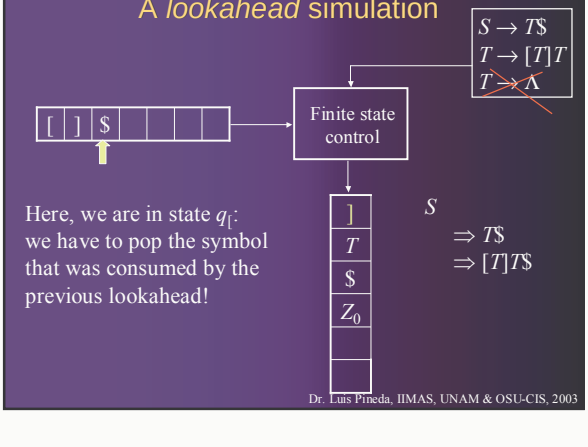
Also, recall the symbol that consumed by the lookahead: Go to state $q_[$

Consume lookahhead symbol

$T \to \Lambda$

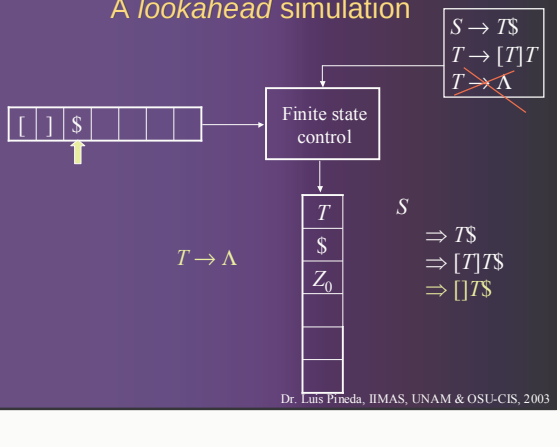
Id	State	Input	Stack symbol	Move(s)
1	q_0	Λ	Z_0	(q_1, SZ_0)
2	q_1	Λ	S	(q_1, TZ_0)
3	q_1	[	T	$(q_[, [T]T)$
4	$q_[$	Λ	[	(q_1, Λ)
5	q_1	]	T	$(q_], \Lambda)$
6	$q_]$	Λ	]	(q_1, Λ)
7	q_1	$	T	$(q_\$, \Lambda)$
8	$q_\$$	Λ	$	(q_1, Λ)
9	q_1	[	[	(q_1, Λ)
10	q_1	]	]	(q_1, Λ)
11	q_1	$	$	(q_1, Λ)
12	q_1	Λ	Z_0	(q_2, Z_0)
Other combinations				non

A *lookahead* simulation

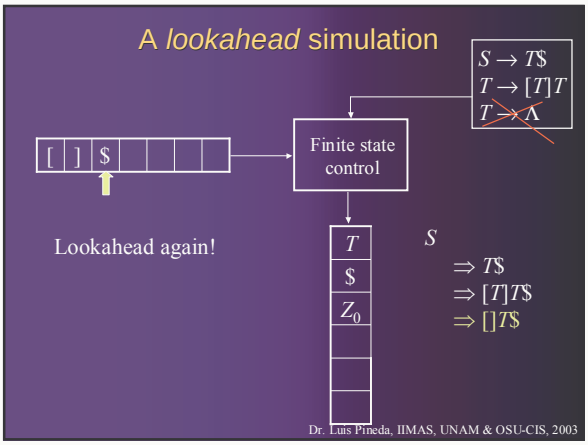


Here, we are in state $q_]$: we have to pop the symbol that was consumed by the previous lookahead!

A *lookahead* simulation



A *lookahead* simulation



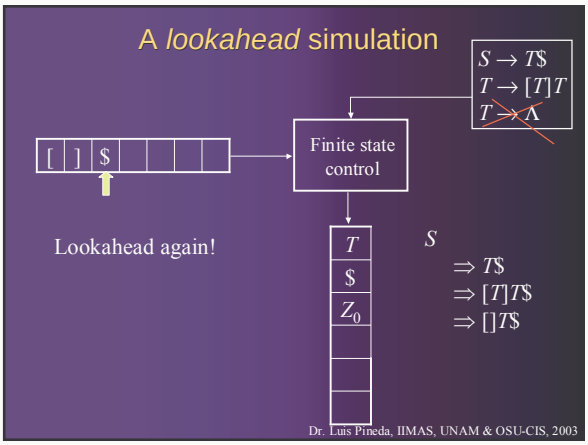
Lookahead again!

lookahhead the $ symbol

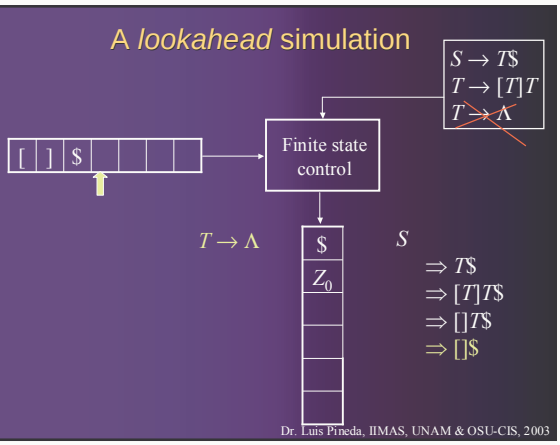
$T \to \Lambda$

Id	State	Input	Stack symbol	Move(s)
1	q_0	Λ	Z_0	(q_1, SZ_0)
2	q_1	Λ	S	(q_1, TZ_0)
3	q_1	[	T	$(q_[, [T]T)$
4	$q_[$	Λ	[	(q_1, Λ)
5	q_1	]	T	$(q_], \Lambda)$
6	$q_]$	Λ	]	(q_1, Λ)
7	q_1	$	T	$(q_\$, \Lambda)$
8	$q_\$$	Λ	$	(q_1, Λ)
9	q_1	[	[	(q_1, Λ)
10	q_1	]	]	(q_1, Λ)
11	q_1	$	$	(q_1, Λ)
12	q_1	Λ	Z_0	(q_2, Z_0)
Other combinations				non

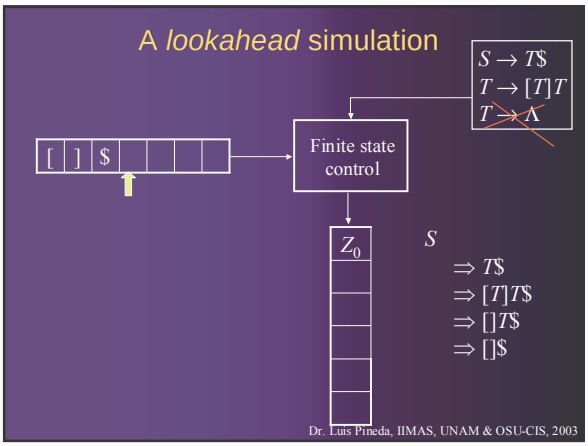
A *lookahead* simulation



A *lookahead* simulation



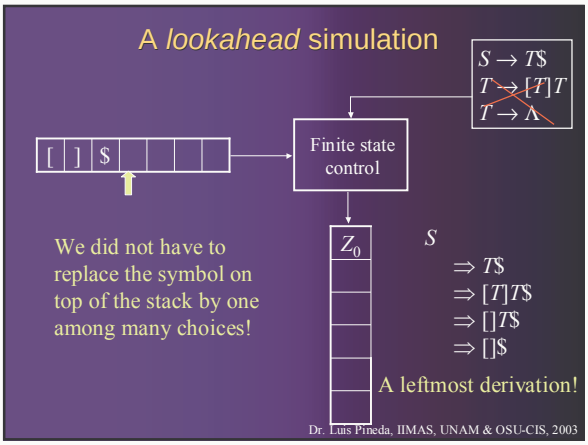
A *lookahead* simulation



deterministic lookahhead PDA

Id	State	Input	Stack symbol	Move(s)
1	q_0	Λ	Z_0	(q_1, SZ_0)
2	q_1	Λ	S	(q_1, TZ_0)
3	q_1	[	T	$(q_1, [T]T)$
4	q_1	Λ	[	(q_1, Λ)
5	q_1	]	T	(q_1, Λ)
6	q_1	Λ	]	(q_1, Λ)
7	q_1	$	T	(q_s, Λ)
8	q_s	Λ	$	(q_1, Λ)
9	q_1	[	[	(q_1, Λ)
10	q_1	]	]	(q_1, Λ)
11	q_1	$	$	(q_1, Λ)
12	q_1	Λ	Z_0	(q_2, Z_0)
Other combinations				non

A *lookahead* simulation



The problem of left-recursion

- Left recursion:
  - A problem with the top-down lookahead simulation:
- Another unambiguous grammar for L

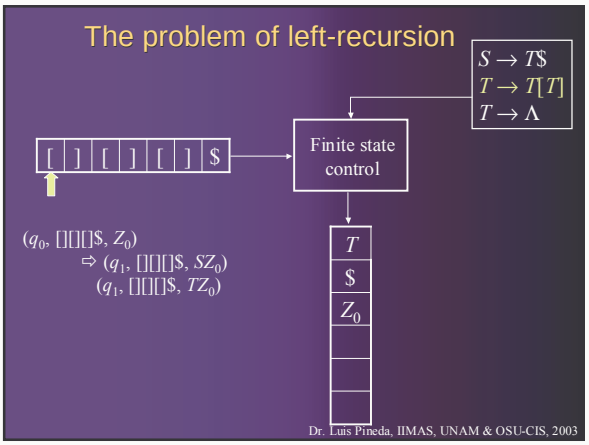
  $S \to T\$$

  $T \to T[T] \mid \Lambda$
- A leftmost derivation:

  $S \Rightarrow T\$ \Rightarrow T[T]\$ \Rightarrow T[T][T]\$ \Rightarrow T[T][T][T]\$\ldots$

  $\ldots\Rightarrow [][][]\$$

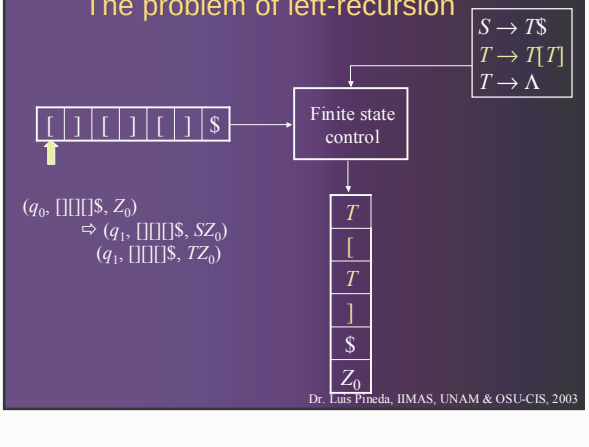
The problem of left-recursion



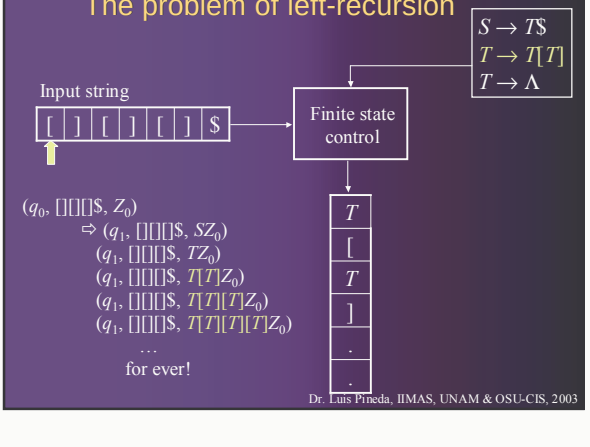
deterministic lookahhead PDA

Id	State	Input	Stack symbol	Move(s)
1	q_0	Λ	Z_0	(q_1, SZ_0)
2	q_1	Λ	S	(q_1, TZ_0)
3	q_1	[	T	$(q_1, T[T])$
4	q_1	Λ	[	(q_1, Λ)
5	q_1	]	T	(q_1, Λ)
6	q_1	Λ	]	(q_1, Λ)
7	q_1	$	T	(q_s, Λ)
8	q_s	Λ	$	(q_1, Λ)
9	q_1	[	[	(q_1, Λ)
10	q_1	]	]	(q_1, Λ)
11	q_1	$	$	(q_1, Λ)
12	q_1	Λ	Z_0	(q_2, Z_0)
Other combinations				non

The problem of left-recursion



The problem of left-recursion



Eliminating left-recursion

- The solution:
  - Find an equivalent grammar without left-recursion
- Consider the T-productions (β does not begin with *T*)

  $T \to T\alpha \mid \beta$
- Replace by

  $T \to \beta U$ and $U \to \alpha U \mid \Lambda$
- Example:

  $T \to T[T] \mid \Lambda \qquad (\alpha = [T], \beta = \Lambda)$
- Replace by

  $T \to U$ and $U \to [T]U \mid \Lambda$

Factoring

- Consider again the balanced parenthesis grammar:

  $S \to T\$$, $T \to [T]T \mid \Lambda$
- Eliminate Λ-productions:

  $S \to T\$$, $T \to [T]T \mid []T \mid [T] \mid []$
- Consider the construction of a lookahead PDA:

  $S \to T\$$, $T \to [T]T \mid []T \mid [T] \mid []$
  - when *T* is on top of the stack there are 4 choices!
- Factor the first symbol of the right sides:

  $S \to T\$$, $T \to [\mathrm{U}$, $\mathrm{U} \to T]T \mid]T \mid T] \mid]$

Factoring

- Right-sides starting with $T]$ can also be factored:
  $S \to T\$,\ T \to [U,\quad U \to T]T \mid]T \mid T] \mid]$
- Factor again rights sides starting with the same symbol
  Let $W \to T \mid \Lambda$
  Then in $U \to T]W \mid]W$
  So
  $S \to T\$,\ T \to [U,\quad U \to T]W \mid]W$ and $W \to T \mid \Lambda$

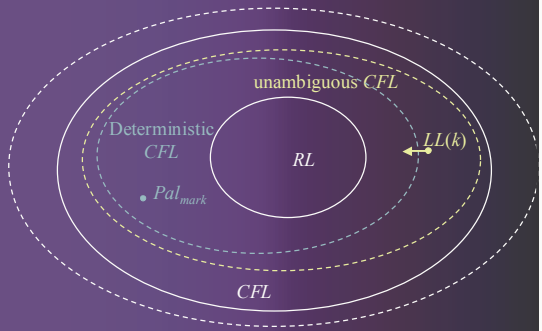
  Λ Λ
- Eliminating the variable T ($T \to [U$):
  $S \to [U\ \$,\ U \to [U]W \mid]W$ and $W \to [U \mid \Lambda$
- The first leftmost symbol of a right-side of productions with the same left-side variable are all different!

LL(1) Grammar

- LL(1) Grammars:
  - Without left-recursion
  - Factoring: For all productions $\alpha \to a_i\beta$, where a_i is different for all productions with left-side α

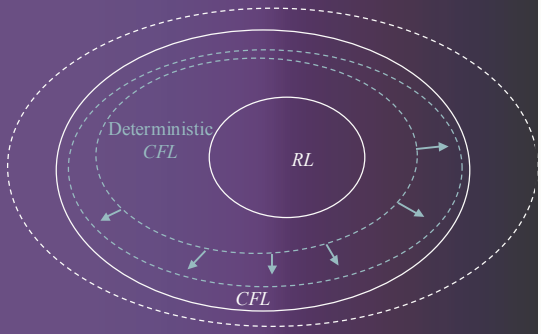
  Can be parsed by a deterministic top-down PDA by looking one symbol ahead in the tape
- LL(k) Grammars:
  - Deterministic PDA
  - Looking k symbols ahead

Deterministic and ambiguous CFL



- Unambiguous CFLs can be parsed deterministically by looking ahead k symbols

Deterministic parsing and DCFL



- Deterministic parsing extends the class of DCFL