Computational Linguistics II: Parsing

LR-Parsing

Frank Richter & Jan-Philipp Söhn

fr@sfs.uni-tuebingen.de, jp.soehn@uni-tuebingen.de

January 29th, 2007

Overview

1. Properties of det. cf. languages
2. Non-Deterministic $\Rightarrow$ Det. FSA
3. LR(1) and ϵ rule
4. LALR(1) Parsing
5. SLR(1) Parsing with JFLAP

Overview

1. Properties of det. cf. languages
2. Non-Deterministic $\Rightarrow$ Det. FSA
3. LR(1) and ϵ rule
4. LALR(1) Parsing
5. SLR(1) Parsing with JFLAP

Overview

1. Properties of det. cf. languages
2. Non-Deterministic $\Rightarrow$ Det. FSA
3. LR(1) and ϵ rule
4. LALR(1) Parsing
5. SLR(1) Parsing with JFLAP

Overview

1. Properties of det. cf. languages
2. Non-Deterministic $\Rightarrow$ Det. FSA
3. LR(1) and ϵ rule
4. LALR(1) Parsing
5. SLR(1) Parsing with JFLAP

Overview

1. Properties of det. cf. languages
2. Non-Deterministic $\Rightarrow$ Det. FSA
3. LR(1) and ϵ rule
4. LALR(1) Parsing
5. SLR(1) Parsing with JFLAP

Once Again: The Big Picture

hierarchy	grammar	machine	other
type 3	reg. grammar	DFA	reg. expressions
		NFA	
det. cf.	LR(k) grammar	DPDA	
type 2	CFG	PDA	
type 1	CSG	LBA	
type 0	unrestricted grammar	Turing machine	

DFA: Deterministic finite state automaton
(D)PDA: (Deterministic) Pushdown automaton
CFG: Context-free grammar
CSG: Context-sensitive grammar
LBA: Linear bounded automaton

Closure Properties

Union

- det. cf. languages are not closed

Concatenation

- det. cf. languages are not closed

Complementation

- det. cf. languages are closed

Kleene star

- det. cf. languages are not closed

Intersection

- det. cf. languages are not closed
- the intersection of a det. cf. language with a regular language is also det. cf.

Decision Properties

Word problem

- all type 2 languages: decidable (CYK algorithm)
- det. cf. languages: linear complexity

Emptiness problem

- all type 2 languages: decidable (marking of symbols in grammar)

Finiteness problem

- all type 2 languages: decidable (cycles in grammar-graph)

Equivalence problem

- det. cf. languages: decidable (proved 1997)

Intersection problem

- det. cf. languages: not decidable (not closed unter intersection)

"What are LR(k) grammars?"

- large subclass of type 2 grammars
- LR grammars are not ambiguous
- matter of definition: a grammar is LR if it can be parsed by an LR parser...
- for a grammar to be LR: recognize a RHS of a production with k input symbols of look-ahead
- for a grammar to be LL: recognize the use of a production seeing only the first k symbols of its RHS.
- thus, LR grammars can describe more languages

"What are LR(k) grammars?"

- large subclass of type 2 grammars
- LR grammars are not ambiguous
- matter of definition: a grammar is LR if it can be parsed by an LR parser...
- for a grammar to be LR: recognize a RHS of a production with k input symbols of look-ahead
- for a grammar to be LL: recognize the use of a production seeing only the first k symbols of its RHS.
- thus, LR grammars can describe more languages

"What are LR(k) grammars?"

- large subclass of type 2 grammars
- LR grammars are not ambiguous
- matter of definition: a grammar is LR if it can be parsed by an LR parser...
- for a grammar to be LR: recognize a RHS of a production with k input symbols of look-ahead
- for a grammar to be LL: recognize the use of a production seeing only the first k symbols of its RHS.
- thus, LR grammars can describe more languages

"What are LR(k) grammars?"

- large subclass of type 2 grammars
- LR grammars are not ambiguous
- matter of definition: a grammar is LR if it can be parsed by an LR parser...
- for a grammar to be LR: recognize a RHS of a production with k input symbols of look-ahead
- for a grammar to be LL: recognize the use of a production seeing only the first k symbols of its RHS.
- thus, LR grammars can describe more languages

"What are LR(k) grammars?"

- large subclass of type 2 grammars
- LR grammars are not ambiguous
- matter of definition: a grammar is LR if it can be parsed by an LR parser...
- for a grammar to be LR: recognize a RHS of a production with k input symbols of look-ahead
- for a grammar to be LL: recognize the use of a production seeing only the first k symbols of its RHS.
- thus, LR grammars can describe more languages

"What are LR(k) grammars?"

- large subclass of type 2 grammars
- LR grammars are not ambiguous
- matter of definition: a grammar is LR if it can be parsed by an LR parser...
- for a grammar to be LR: recognize a RHS of a production with k input symbols of look-ahead
- for a grammar to be LL: recognize the use of a production seeing only the first k symbols of its RHS.
- thus, LR grammars can describe more languages

Prerequisite: Non-Deterministic $\Rightarrow$ Det. FSA

- recognition with a **non-deterministic FSA** is very inefficient: involves extensive search
- at every point when different transitions are possible, try both alternatives
- solution: convert **non-deterministic FSA** into **deterministic FSA**
- recognized language must remain the same!
- two steps:
  1. subset construction
  2. reconnecting states
- deterministic algorithms generally have more states, ca. $10 \times n$

Prerequisite: Non-Deterministic ⇒ Det. FSA

- recognition with a **non-deterministic FSA** is very inefficient: involves extensive search
- at every point when different transitions are possible, try both alternatives
- solution: convert **non-deterministic FSA** into **deterministic FSA**
- recognized language must remain the same!
- two steps:
  1. subset construction
  2. reconnecting states
- deterministic algorithms generally have more states, ca. $10 \times n$

Prerequisite: Non-Deterministic $\Rightarrow$ Det. FSA

- recognition with a **non-deterministic FSA** is very inefficient: involves extensive search
- at every point when different transitions are possible, try both alternatives
- solution: convert **non-deterministic FSA** into **deterministic FSA**
- recognized language must remain the same!
- two steps:
  1. subset construction
  2. reconnecting states
- deterministic algorithms generally have more states, ca. $10 \times n$

Prerequisite: Non-Deterministic ⇒ Det. FSA

- recognition with a **non-deterministic FSA** is very inefficient: involves extensive search
- at every point when different transitions are possible, try both alternatives
- solution: convert **non-deterministic FSA** into **deterministic FSA**
- recognized language must remain the same!
- two steps:
  1. subset construction
  2. reconnecting states
- deterministic algorithms generally have more states, ca. $10 \times n$

Prerequisite: Non-Deterministic $\Rightarrow$ Det. FSA

- recognition with a **non-deterministic FSA** is very inefficient: involves extensive search
- at every point when different transitions are possible, try both alternatives
- solution: convert **non-deterministic FSA** into **deterministic FSA**
- recognized language must remain the same!
- two steps:
  1. subset construction
  2. reconnecting states
- deterministic algorithms generally have more states, ca. $10 \times n$

Prerequisite: Non-Deterministic ⇒ Det. FSA

- recognition with a **non-deterministic FSA** is very inefficient: involves extensive search
- at every point when different transitions are possible, try both alternatives
- solution: convert **non-deterministic FSA** into **deterministic FSA**
- recognized language must remain the same!
- two steps:
  1. subset construction
  2. reconnecting states
- deterministic algorithms generally have more states, ca. $10 \times n$

Prerequisite: Non-Deterministic ⇒ Det. FSA

- recognition with a **non-deterministic FSA** is very inefficient: involves extensive search
- at every point when different transitions are possible, try both alternatives
- solution: convert **non-deterministic FSA** into **deterministic FSA**
- recognized language must remain the same!
- two steps:
  1. subset construction
  2. reconnecting states
- deterministic algorithms generally have more states, ca. $10 \times n$

Prerequisite: Non-Deterministic ⇒ Det. FSA

- recognition with a **non-deterministic FSA** is very inefficient: involves extensive search
- at every point when different transitions are possible, try both alternatives
- solution: convert **non-deterministic FSA** into **deterministic FSA**
- recognized language must remain the same!
- two steps:
  1. subset construction
  2. reconnecting states
- deterministic algorithms generally have more states, ca. $10 \times n$

Subset Construction

- old start state = new start state
- constructing a state tree:
  for each new state s in the new automaton:
    for each element e in the lexicon:
      create a new state x which is the subset of all states that can be reached from s via e
      create a transition from s to x with label e
  newly created states which already exist receive a mark but are not pursued further
- result: a deterministic state tree

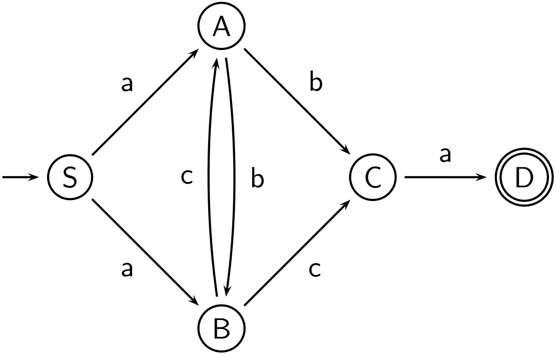
Subset Construction

- old start state = new start state
- constructing a state tree:
  for each new state s in the new automaton:
  for each element e in the lexicon:
  create a new state x which is the subset of all states that can be reached from s via e
  create a transition from s to x with label e
  newly created states which already exist receive a mark but are not pursued further
- result: a deterministic state tree

Subset Construction

- old start state = new start state
- constructing a state tree:
  for each new state s in the new automaton:
  for each element e in the lexicon:
  create a new state x which is the subset of all states that can be reached from s via e
  create a transition from s to x with label e
  newly created states which already exist receive a mark but are not pursued further
- result: a deterministic state tree

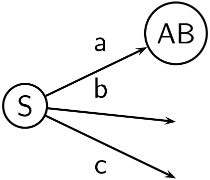
Example



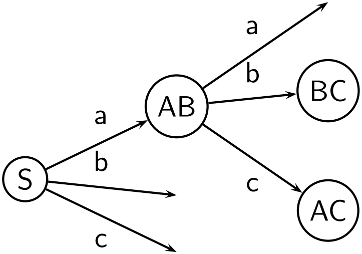
Example

S

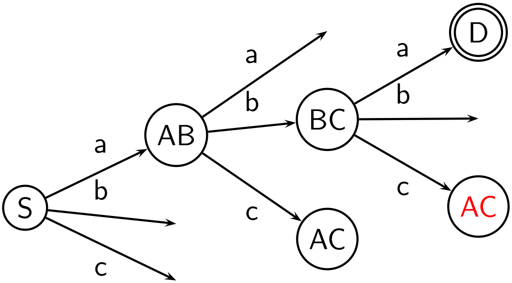
Example



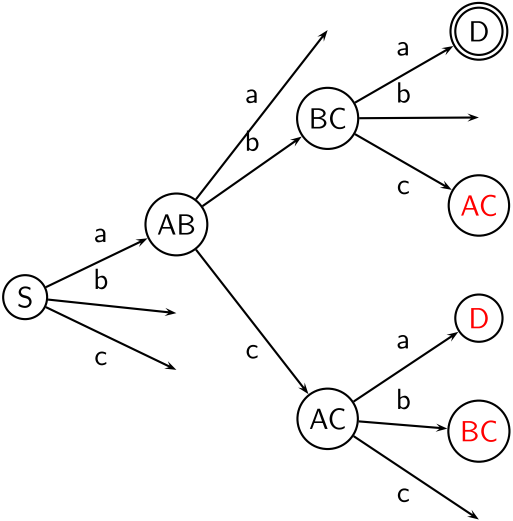
Example



Example

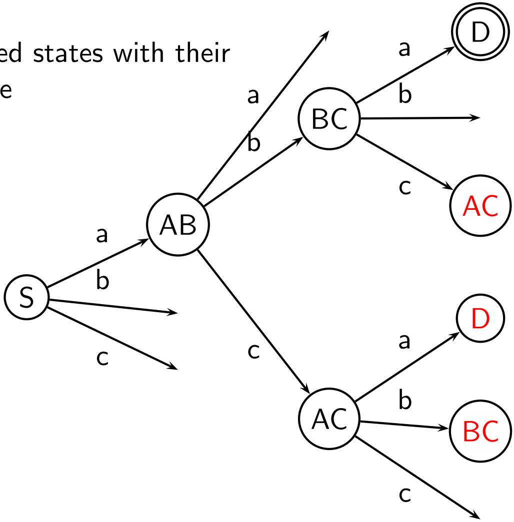


Example



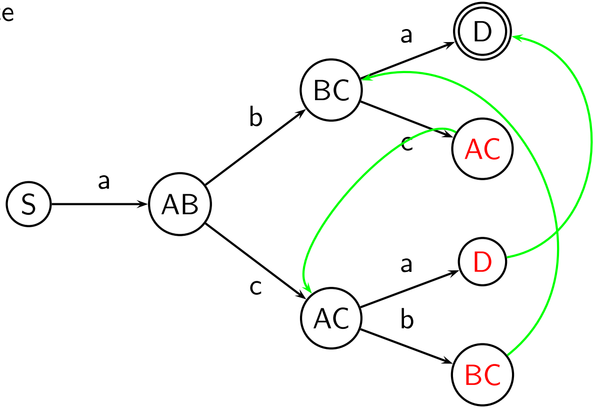
Reconnecting the Automaton

- delete transitions which lead to error states
- combine marked states with their first occurrence

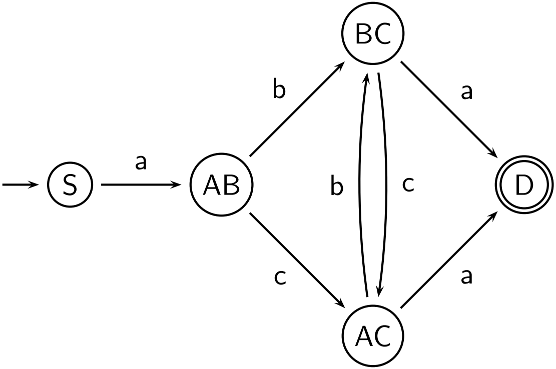


Reconnecting the Automaton

- delete transitions which lead to error states
- combine marked states with their first occurrence



Reconnecting the Automaton



LR(1) and ϵ rules

- we know that ϵ rules are difficult for bottom-up parsers: can be inserted anywhere between words, any number of times
- in non-deterministic automaton: no problem, just like any other rule
- in deterministic automaton: only works when look-ahead is different from any other rule in the same state
- otherwise, a shift/reduce or a reduce/reduce conflict results
- one needs to be careful when constructing look-ahead sets: category that dominates ϵ is "transparent"
- S $\rightarrow$ ABC; A $\rightarrow$ a; B $\rightarrow$ ϵ | b; C $\rightarrow$ c
  FOLLOW(A)= {b, c}
- the presence of ϵ rules in a grammar reduces the likelihood of the grammar to be LR(1)

LR(1) and ϵ rules

- we know that ϵ rules are difficult for bottom-up parsers: can be inserted anywhere between words, any number of times
- in non-deterministic automaton: no problem, just like any other rule
- in deterministic automaton: only works when look-ahead is different from any other rule in the same state
- otherwise, a shift/reduce or a reduce/reduce conflict results
- one needs to be careful when constructing look-ahead sets: category that dominates ϵ is "transparent"
- S $\rightarrow$ ABC; A $\rightarrow$ a; B $\rightarrow$ ϵ | b; C $\rightarrow$ c
  FOLLOW(A)= {b, c}
- the presence of ϵ rules in a grammar reduces the likelihood of the grammar to be LR(1)

LR(1) and ϵ rules

- we know that ϵ rules are difficult for bottom-up parsers: can be inserted anywhere between words, any number of times
- in non-deterministic automaton: no problem, just like any other rule
- in deterministic automaton: only works when look-ahead is different from any other rule in the same state
- otherwise, a shift/reduce or a reduce/reduce conflict results
- one needs to be careful when constructing look-ahead sets: category that dominates ϵ is "transparent"
- S $\rightarrow$ ABC; A $\rightarrow$ a; B $\rightarrow$ ϵ | b; C $\rightarrow$ c
  FOLLOW(A)= {b, c}
- the presence of ϵ rules in a grammar reduces the likelihood of the grammar to be LR(1)

LR(1) and ϵ rules

- we know that ϵ rules are difficult for bottom-up parsers: can be inserted anywhere between words, any number of times
- in non-deterministic automaton: no problem, just like any other rule
- in deterministic automaton: only works when look-ahead is different from any other rule in the same state
- otherwise, a shift/reduce or a reduce/reduce conflict results
- one needs to be careful when constructing look-ahead sets: category that dominates ϵ is "transparent"
- S $\rightarrow$ ABC; A $\rightarrow$ a; B $\rightarrow$ ϵ | b; C $\rightarrow$ c
  FOLLOW(A)= {b, c}
- the presence of ϵ rules in a grammar reduces the likelihood of the grammar to be LR(1)

LR(1) and ϵ rules

- we know that ϵ rules are difficult for bottom-up parsers: can be inserted anywhere between words, any number of times
- in non-deterministic automaton: no problem, just like any other rule
- in deterministic automaton: only works when look-ahead is different from any other rule in the same state
- otherwise, a shift/reduce or a reduce/reduce conflict results
- one needs to be careful when constructing look-ahead sets: category that dominates ϵ is "transparent"
- S $\rightarrow$ ABC; A $\rightarrow$ a; B $\rightarrow$ ϵ | b; C $\rightarrow$ c
  FOLLOW(A)= {b, c}
- the presence of ϵ rules in a grammar reduces the likelihood of the grammar to be LR(1)

LR(1) and ϵ rules

- we know that ϵ rules are difficult for bottom-up parsers: can be inserted anywhere between words, any number of times
- in non-deterministic automaton: no problem, just like any other rule
- in deterministic automaton: only works when look-ahead is different from any other rule in the same state
- otherwise, a shift/reduce or a reduce/reduce conflict results
- one needs to be careful when constructing look-ahead sets: category that dominates ϵ is "transparent"
- S $\rightarrow$ ABC; A $\rightarrow$ a; B $\rightarrow$ ϵ | b; C $\rightarrow$ c
  FOLLOW(A)= {b, c}
- the presence of ϵ rules in a grammar reduces the likelihood of the grammar to be LR(1)

LR(1) and ϵ rules

- we know that ϵ rules are difficult for bottom-up parsers: can be inserted anywhere between words, any number of times
- in non-deterministic automaton: no problem, just like any other rule
- in deterministic automaton: only works when look-ahead is different from any other rule in the same state
- otherwise, a shift/reduce or a reduce/reduce conflict results
- one needs to be careful when constructing look-ahead sets: category that dominates ϵ is "transparent"
- S $\rightarrow$ ABC; A $\rightarrow$ a; B $\rightarrow$ ϵ | b; C $\rightarrow$ c
  FOLLOW(A)= {b, c}
- the presence of ϵ rules in a grammar reduces the likelihood of the grammar to be LR(1)

FIRST and FOLLOW

The FIRST set

1. FIRST(ϵ) $= \epsilon$
2. FIRST(a) $= a$
3. FIRST(A) is the union of FIRST(w) for all RHS w of A.
4. Let every X_i be either a terminal or a variable:
   FIRST($X_1X_2X_3...X_N$) = FIRST(X_1) if X_1 does not derive ϵ
   FIRST($X_1X_2X_3...X_N$) = FIRST(X_1) $- \epsilon \cup$ FIRST($X_2X_3...X_N$) if X_1 derives ϵ

The FOLLOW set

1. $\#$ is in FOLLOW(S)
2. for A→vB, FOLLOW(A) is in FOLLOW(B).
3. for A→vBw:
   FIRST(w) $- \epsilon$ is in FOLLOW(B)
   if $\epsilon \in$ FIRST(w), then FOLLOW(A) is in FOLLOW(B)

FIRST and FOLLOW

The FIRST set

1. FIRST(ϵ) $= \epsilon$
2. FIRST(a) $= a$
3. FIRST(A) is the union of FIRST(w) for all RHS w of A.
4. Let every X_i be either a terminal or a variable:
   FIRST($X_1 X_2 X_3 \ldots X_N$) = FIRST(X_1) if X_1 does not derive ϵ
   FIRST($X_1 X_2 X_3 \ldots X_N$) = FIRST(X_1) $- \epsilon \cup$ FIRST($X_2 X_3 \ldots X_N$) if X_1 derives ϵ

The FOLLOW set

1. $\#$ is in FOLLOW(S)
2. for A$\rightarrow$vB, FOLLOW(A) is in FOLLOW(B).
3. for A$\rightarrow$vBw:
   FIRST(w) $- \epsilon$ is in FOLLOW(B)
   if $\epsilon \in$ FIRST(w), then FOLLOW(A) is in FOLLOW(B)

FIRST and FOLLOW

The FIRST set

1. FIRST(ϵ) $= \epsilon$
2. FIRST(a) $= a$
3. FIRST(A) is the union of FIRST(w) for all RHS w of A.
4. Let every X_i be either a terminal or a variable:
   FIRST($X_1 X_2 X_3 ... X_N$) = FIRST(X_1) if X_1 does not derive ϵ
   FIRST($X_1 X_2 X_3 ... X_N$) = FIRST(X_1) $- \epsilon \cup$ FIRST($X_2 X_3 ... X_N$) if X_1 derives ϵ

The FOLLOW set

1. # is in FOLLOW(S)
2. for A→vB, FOLLOW(A) is in FOLLOW(B).
3. for A→vBw:
   FIRST(w) $- \epsilon$ is in FOLLOW(B)
   if $\epsilon \in$ FIRST(w), then FOLLOW(A) is in FOLLOW(B)

FIRST and FOLLOW

The FIRST set

1. FIRST(ϵ) $= \epsilon$
2. FIRST(a) $= a$
3. FIRST(A) is the union of FIRST(w) for all RHS w of A.
4. Let every X_i be either a terminal or a variable:
   FIRST($X_1 X_2 X_3 ... X_N$) $=$ FIRST(X_1) if X_1 does not derive ϵ
   FIRST($X_1 X_2 X_3 ... X_N$) $=$ FIRST(X_1) $- \epsilon \cup$ FIRST($X_2 X_3 ... X_N$) if X_1 derives ϵ

The FOLLOW set

1. $\#$ is in FOLLOW(S)
2. for A$\rightarrow$vB, FOLLOW(A) is in FOLLOW(B).
3. for A$\rightarrow$vBw:
   FIRST(w) $- \epsilon$ is in FOLLOW(B)
   if $\epsilon \in$ FIRST(w), then FOLLOW(A) is in FOLLOW(B)

FIRST and FOLLOW

The FIRST set

1. FIRST(ϵ) $= \epsilon$
2. FIRST(a) $= a$
3. FIRST(A) is the union of FIRST(w) for all RHS w of A.
4. Let every X_i be either a terminal or a variable:
   FIRST($X_1 X_2 X_3 ... X_N$) $=$ FIRST(X_1) if X_1 does not derive ϵ
   FIRST($X_1 X_2 X_3 ... X_N$) $=$ FIRST(X_1) $- \epsilon \cup$ FIRST($X_2 X_3 ... X_N$) if X_1 derives ϵ

The FOLLOW set

1. $\#$ is in FOLLOW(S)
2. for A$\rightarrow$vB, FOLLOW(A) is in FOLLOW(B).
3. for A$\rightarrow$vBw:
   FIRST(w) $- \epsilon$ is in FOLLOW(B)
   if $\epsilon \in$ FIRST(w), then FOLLOW(A) is in FOLLOW(B)

FIRST and FOLLOW

The FIRST set

1. FIRST(ϵ) $= \epsilon$
2. FIRST(a) $= a$
3. FIRST(A) is the union of FIRST(w) for all RHS w of A.
4. Let every X_i be either a terminal or a variable:
   FIRST($X_1 X_2 X_3 \ldots X_N$) = FIRST(X_1) if X_1 does not derive ϵ
   FIRST($X_1 X_2 X_3 \ldots X_N$) = FIRST(X_1) $- \epsilon \cup$ FIRST($X_2 X_3 \ldots X_N$) if X_1 derives ϵ

The FOLLOW set

1. $\#$ is in FOLLOW(S)
2. for A$\longrightarrow$vB, FOLLOW(A) is in FOLLOW(B).
3. for A$\longrightarrow$vBw:
   FIRST(w) $- \epsilon$ is in FOLLOW(B)
   if $\epsilon \in$ FIRST(w), then FOLLOW(A) is in FOLLOW(B)

FIRST and FOLLOW

The FIRST set

1. FIRST(ϵ) $= \epsilon$
2. FIRST(a) $= a$
3. FIRST(A) is the union of FIRST(w) for all RHS w of A.
4. Let every X_i be either a terminal or a variable:
   FIRST($X_1 X_2 X_3 ... X_N$) = FIRST(X_1) if X_1 does not derive ϵ
   FIRST($X_1 X_2 X_3 ... X_N$) = FIRST(X_1) $- \epsilon \cup$ FIRST($X_2 X_3 ... X_N$) if X_1 derives ϵ

The FOLLOW set

1. $\#$ is in FOLLOW(S)
2. for A$\longrightarrow$vB, FOLLOW(A) is in FOLLOW(B).
3. for A$\longrightarrow$vBw:
   FIRST(w) $- \epsilon$ is in FOLLOW(B)
   if $\epsilon \in$ FIRST(w), then FOLLOW(A) is in FOLLOW(B)

LALR(1) Parsing

- problem with LR(1) parsing: huge tables
- idea: go back to LR(0) table: LR(1) states can be collapsed to LR(0) states without changing transitions
- only missing information: look-aheads!
  Can be copied from LR(1) states.
- result: Look Ahead LR(0) with 1 look-ahead: LALR(1)
- BUT: need to construct huge LR(1) automaton first!
- needed: technique to insert look-ahead information without LR(1) automaton

LALR(1) Parsing

- problem with LR(1) parsing: huge tables
- idea: go back to LR(0) table: LR(1) states can be collapsed to LR(0) states without changing transitions
- only missing information: look-aheads!
  Can be copied from LR(1) states.
- result: Look Ahead LR(0) with 1 look-ahead: LALR(1)
- BUT: need to construct huge LR(1) automaton first!
- needed: technique to insert look-ahead information without LR(1) automaton

LALR(1) Parsing

- problem with LR(1) parsing: huge tables
- idea: go back to LR(0) table: LR(1) states can be collapsed to LR(0) states without changing transitions
- only missing information: look-aheads!
  Can be copied from LR(1) states.
- result: Look Ahead LR(0) with 1 look-ahead: LALR(1)
- BUT: need to construct huge LR(1) automaton first!
- needed: technique to insert look-ahead information without LR(1) automaton

LALR(1) Parsing

- problem with LR(1) parsing: huge tables
- idea: go back to LR(0) table: LR(1) states can be collapsed to LR(0) states without changing transitions
- only missing information: look-aheads!
  Can be copied from LR(1) states.
- result: Look Ahead LR(0) with 1 look-ahead: LALR(1)
- BUT: need to construct huge LR(1) automaton first!
- needed: technique to insert look-ahead information without LR(1) automaton

LALR(1) Parsing

- problem with LR(1) parsing: huge tables
- idea: go back to LR(0) table: LR(1) states can be collapsed to LR(0) states without changing transitions
- only missing information: look-aheads!
  Can be copied from LR(1) states.
- result: Look Ahead LR(0) with 1 look-ahead: LALR(1)
- BUT: need to construct huge LR(1) automaton first!
- needed: technique to insert look-ahead information without LR(1) automaton

LALR(1) Parsing

- problem with LR(1) parsing: huge tables
- idea: go back to LR(0) table: LR(1) states can be collapsed to LR(0) states without changing transitions
- only missing information: look-aheads!
  Can be copied from LR(1) states.
- result: Look Ahead LR(0) with 1 look-ahead: LALR(1)
- BUT: need to construct huge LR(1) automaton first!
- needed: technique to insert look-ahead information without LR(1) automaton

Channel Algorithm

- see Sandra's slides

SLR(1)

A simpler way:

- Extract the FOLLOW sets from the grammar
- construct LR(0) automaton
- add look-aheads for each item A→... according to FOLLOW(A)

SLR(1)

A simpler way:

- Extract the FOLLOW sets from the grammar
- construct LR(0) automaton
- add look-aheads for each item A→... according to FOLLOW(A)

SLR(1)

A simpler way:

- Extract the FOLLOW sets from the grammar
- construct LR(0) automaton
- add look-aheads for each item A→... according to FOLLOW(A)