TDDD55 Compilers and Interpreters
TDDB44 Compiler Construction

LR Parsing, Part 2

Constructing Parse Tables

Parse table construction

Grammar conflict handling

Categories of LR Grammars and Parsers

Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2011.

Need to Automatically Construct LR Parse Tables: Action and GOTO Table

Construct parse tables from the grammar as follows:

- First build a GOTOgraph (an NFA) to recognize viable prefixes
- Make it deterministic (DFA)
- Then fill in Action and Goto tables

ACTION table:

state	|--	,	**a**	**b**
0	X	X	S4	S5
1	A	S2	*	*
2	X	X	S4	S5
3	R1	R1	*	*
4	R3	R3	*	*
5	R4	R4	*	*
6				

GOTO table:

state	L	E
0	1	6
1	*	*
2	*	3
3	*	*
4	*	*
5	*	*
6	*	*

Example Grammar G

1. <L → L , E
2. | E
3. E → a
4. | b

Classes of LR Parsers/Grammars

- LR(0) – Too weak (no lookahead)
- SLR(1) – *Simple LR, 1 token lookahead*
- LALR(1) – Most common, 1 token lookahead
- LR(1) – 1 token lookahead – big tables
- LR(k) – k tokens lookahead – Even bigger tables

Differences between LR parsers:

- Table size varies widely.
- Errors not discovered as quickly by some variants.
- Different limitations in the language definitions, grammars.

118a

An NFA Recognizing Viable Prefixes

A.k.a. the "*characteristic finite automaton*" for a grammar G

- States: LR(0) items (= context-free items) of extended Grammar (definition, see next page)
- Input stream: The grammar symbols on the stack
- Start state: [S' → –|.S] Final state: [S' → –|S.]
- Transitions:
  - "**move dot across symbol**" if symbol found next on stack:
    $A \to \alpha.B\gamma$ to $A \to \alpha B.\gamma$
    $A \to \alpha.b\gamma$ to $A \to \alpha b.\gamma$
  - **ε-transitions** to LR(0)-items for nonterminal productions from items where the dot precedes that nonterminal:
    $A \to \alpha.B\gamma$ to $B \to .\beta$

Handle, Viable Prefix

- Consider a rightmost derivation $S =>^{*}_{rm} \beta X u =>_{rm} \beta\alpha u$ for a context-free grammar *G*.
- α is called a **handle** of the right sentential form $\beta\alpha u$, associated with the rule $X =>_{rm} \alpha$
- Each prefix of $\beta\alpha$ is called a **viable prefix** of *G*.

Example: Grammar *G* with productions { S -> aSb | c }

- Right sentential forms: e.g. c, acb, aSb, aaaaaSbbbbb, ...
- For c: Handle: c Viable prefixes: ε, c
- For acb: Handle: c ε, a, ac
- For aSb: Handle: aSb ε, a, aS, aSb
- For aaSbb: Handle: aSb ε, a, aa, aaS, aaSb
- ...

Right Derivation and Viable Prefixes

Input: a, b, a

Right derivation (handles are underlined, and blue)

<list> $=>_{rm}$ <list> , <element>
$=>_{rm}$ <list> , a
$=>_{rm}$ <list> , <element> , a
$=>_{rm}$ <list> , b , a
$=>_{rm}$ <element> , b, a
$=>_{rm}$ a , b, a

Some Viable prefixes of the sentential form: <list> , b, a

are

{ ε; <list> ; <list>, ; <list>, b ; <list>, b , ; <list>, b , a }

Definition of LR(0) Item

- An *LR(0) item of a rule P is a rule with a dot "•" somewhere in* the right side.

Example:

- All LR(0) items of the production
  1. <list> → <list> , <element>

are

<list> → • <list> , <element>
<list> → <list> • , <element>
<list> → <list> , • <element>
<list> → <list> , <element> •

- Intuitively an *item is interpreted as how much of the rule* we have found and how much remains.
- ***Items are put together in sets which become the LR*** analyser's state.

Informal Construction of GOTO-Graph (NFA/DFA)

We want to construct a DFA which recognises all ***viable prefixes*** *of G(<SYS>):*

GOTO-graph
(A GOTO-graph is **not** the same as a GOTO-table but corresponds to an **ACTION + GOTO-table**.

The graph discovers *viable prefixes.)*

Augmented Grammar G(<sys>)
0. <SYS> → <list> |–
1. <list> → <list> , <element>
2. | <element>
3. <element> → a
4. | b

Example. Find viable prefixes in a rightmost derivation below, used for informal construction of a goto graph

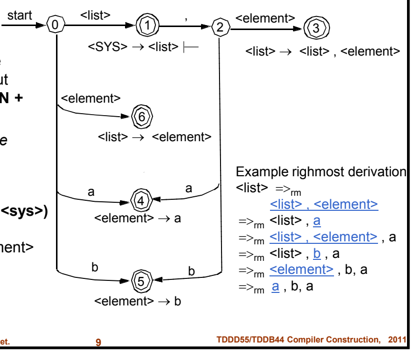
<list> $=>_{rm}$ <list> , <element>
$=>_{rm}$ <list> , a
$=>_{rm}$ <list> , <element> , a
$=>_{rm}$ <list> , b , a
$=>_{rm}$ <element> , b, a
$=>_{rm}$ a , b, a

Informal Construction of GOTO-Graph (NFA/DFA)

We want to construct a DFA which recognises all *viable prefixes of G(<SYS>):*

GOTO-graph
(A GOTO-graph is **not** the same as a GOTO-table but corresponds to an **ACTION + GOTO-table**.

The graph discovers *viable prefixes.)*



Augmented Grammar G(<sys>)
0. <SYS> → <list> |–
1. <list> → <list> , <element>
2. | <element>
3. <element> → a
4. | b

Example righmost derivation
<list> $=>_{rm}$
<list> , <element>
$=>_{rm}$ <list> , a
$=>_{rm}$ <list> , <element> , a
$=>_{rm}$ <list> , b , a
$=>_{rm}$ <element> , b, a
$=>_{rm}$ a , b, a

Constructing Sets of LR(0) Items

Set I_0	<SYS> → • <list> |--	Kernel (Basis)
	<list> → • <list> , <element> <list> → • <element> <element> → • a <element> → • b	Additional Closure (of kernel items)
Set I_1	<SYS> → <list> • |-- <list> → <list> • , <element>	Kernel (Basis)
	(empty *closure as "•" precedes terminals* |-- and , *)*	Additional Closure
Set I_2	<list> → <list> , • <element>	Kernel (Basis)
	<element> → • a <element> → • b	Additional Closure

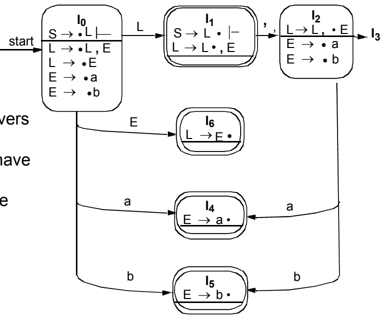
Set I_3, etc.

Augmented Grammar G(<sys>)

0. <SYS>→ <list> |–
1. <list> → <list , <element>
2. | <element>
3. <element> → a
4. | b

GOTO Graph with States as Sets of LR(0) Items

Based on the canonical collection of LR(0) items draw the GOTO graph.

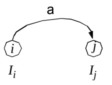


The GOTO graph discovers those prefixes of right sentential forms which have (at most) one handle furthest to the right in the prefix.

Example Grammar
1. L → L , E
2. L → E
3. E → a
4. E → b

Fill in Action Table from GOTO Graph

1. If there is an item
<A> → α • a β ∈ I_i
and
GOTOgraph(I_i , *a)* = I_j



Fill in ***shift j*** *for row* ***i*** *and column for symbol* ***a****.*

2. If there is a complete item (i.e., ends in a dot "•"):
<A> → α• ∈ I_i
Fill in ***reduce x*** *where x is the production number for x: <A> → α*

I_i *: state i (line i, itemset i)*

Filled in Action table

state	|--	,	a	b
0	X	X	S4	S5
1	A	S2	*	*
2	X	X	S4	S5
3	R1	R1	*	*
4	R3	R3	*	*
5	R4	R4	*	*
6	R2	R2	*	*

ACTION table:

State number		a	Nonterminals
i		*shift j*	

3. If we have
<SYS> → <S> • |--
accept the symbol |--

4. Otherwise *error.*

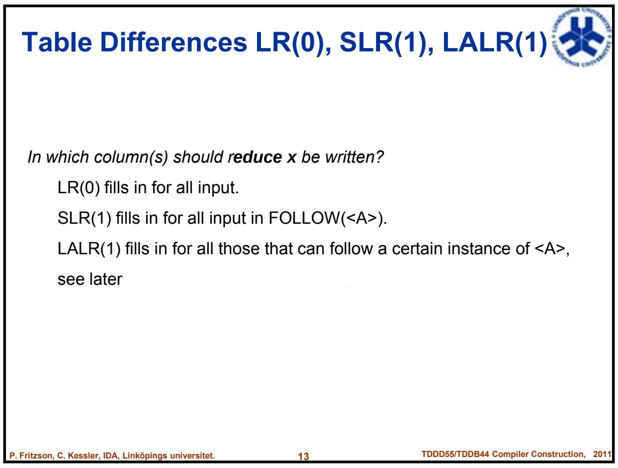


Table Differences LR(0), SLR(1), LALR(1)

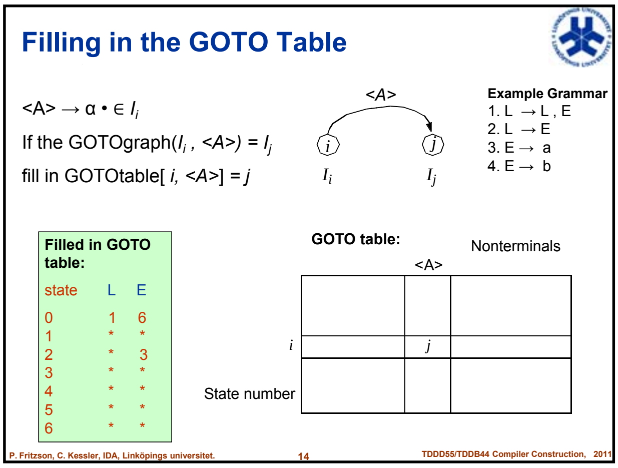
*In which column(s) should **reduce x** be written?*

LR(0) fills in for all input.

SLR(1) fills in for all input in FOLLOW(<A>).

LALR(1) fills in for all those that can follow a certain instance of <A>, see later

---



Filling in the GOTO Table

$\langle A\rangle \to \alpha \bullet \in I_i$

If the GOTOgraph(I_i , <A>) = I_j

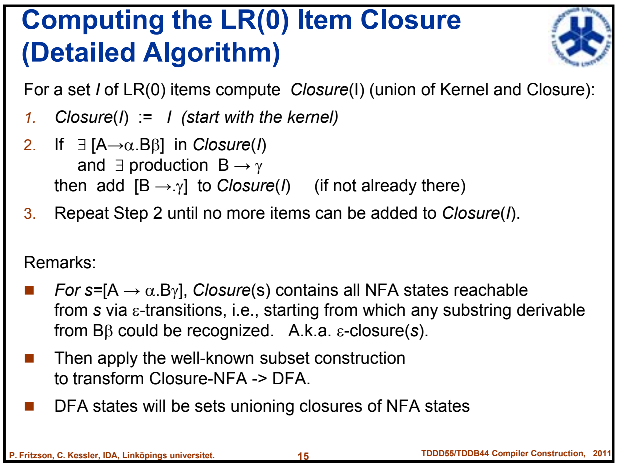
fill in GOTOtable[*i*, <A>] = *j*

Example Grammar
1. L → L , E
2. L → E
3. E → a
4. E → b

Filled in GOTO table:

state	L	E
0	1	6
1	*	*
2	*	3
3	*	*
4	*	*
5	*	*
6	*	*

GOTO table:



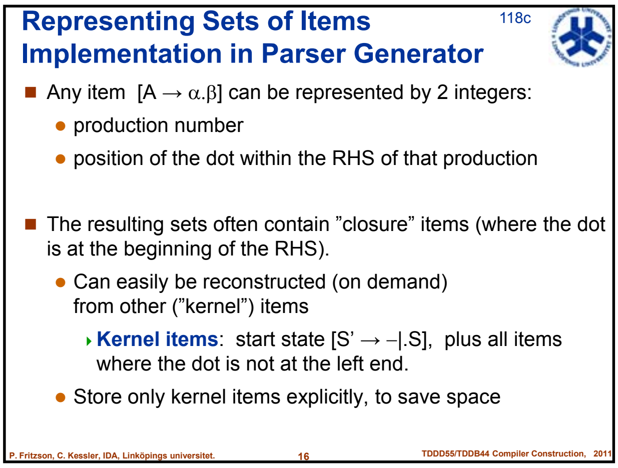
Computing the LR(0) Item Closure (Detailed Algorithm)

For a set *I* of LR(0) items compute *Closure*(I) (union of Kernel and Closure):

1. *Closure*(*I*) := *I* *(start with the kernel)*
2. If ∃ [A→α.Bβ] in *Closure*(*I*)
   and ∃ production B → γ
   then add [B →.γ] to *Closure*(*I*) (if not already there)
3. Repeat Step 2 until no more items can be added to *Closure*(*I*).

Remarks:

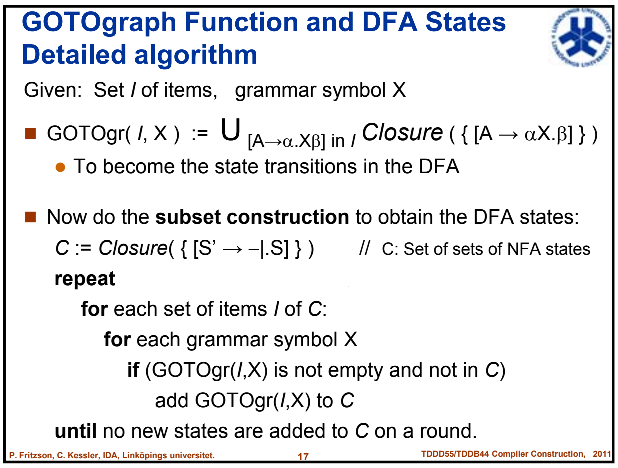
- *For s*=[A → α.Bγ], *Closure*(s) contains all NFA states reachable from *s* via ε-transitions, i.e., starting from which any substring derivable from Bβ could be recognized. A.k.a. ε-closure(*s*).
- Then apply the well-known subset construction to transform Closure-NFA -> DFA.
- DFA states will be sets unioning closures of NFA states



118c

Representing Sets of Items Implementation in Parser Generator

- Any item [A → α.β] can be represented by 2 integers:
  - production number
  - position of the dot within the RHS of that production
- The resulting sets often contain ”closure” items (where the dot is at the beginning of the RHS).
  - Can easily be reconstructed (on demand) from other (”kernel”) items
    - **Kernel items**: start state [S’ → –|.S], plus all items where the dot is not at the left end.
  - Store only kernel items explicitly, to save space

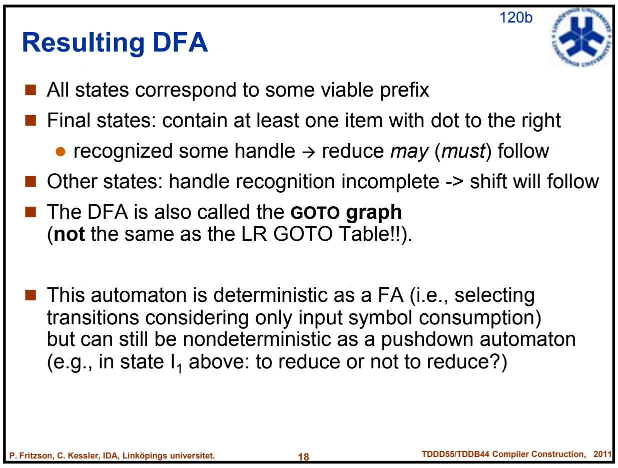


GOTOgraph Function and DFA States Detailed algorithm

Given: Set *I* of items, grammar symbol X

- GOTOgr(*I*, X) := $\bigcup_{[A\to\alpha.X\beta] \text{ in } I}$ *Closure* ({ [A → αX.β] })
  - To become the state transitions in the DFA
- Now do the **subset construction** to obtain the DFA states:

```
C := Closure( { [S' → –|.S] } )      // C: Set of sets of NFA states
repeat
    for each set of items I of C:
        for each grammar symbol X
            if (GOTOgr(I,X) is not empty and not in C)
                add GOTOgr(I,X) to C
until no new states are added to C on a round.
```



120b

Resulting DFA

- All states correspond to some viable prefix
- Final states: contain at least one item with dot to the right
  - recognized some handle → reduce *may* (*must*) follow
- Other states: handle recognition incomplete -> shift will follow
- The DFA is also called the **GOTO graph** (**not** the same as the LR GOTO Table!!).
- This automaton is deterministic as a FA (i.e., selecting transitions considering only input symbol consumption) but can still be nondeterministic as a pushdown automaton (e.g., in state I_1 above: to reduce or not to reduce?)

From DFA to parser tables: ACTION Detailed Algorithm, Summary

1. For each DFA transition $I_i \rightarrow I_j$ reading a terminal a in Σ

   (thus, I_i contains items of kind $[X \rightarrow \alpha.a\beta]$)

   - enter S j (shift, new state I_j) in ACTION[i, a]

2. For each DFA final state I_i
   (containing a complete item $[X \rightarrow \alpha.]$)

   - enter R x
     (reduce, x = prod. rule number for $X \rightarrow \alpha$)
     in ACTION[i, t] ...
     - LR(0) parser: for *all* t in Σ (all entries in row i)
     - SLR(1) parser: for all t in $LA_{SLR}(i,[X \rightarrow \alpha.]) = FOLLOW_1(X)$
     - LALR(1) parser: for all t in $LA_{LALR}(i,[X \rightarrow \alpha.])$ (see later)
   - Collision with an already existing S or R entry? Conflict!!

3. For each DFA state containing $[S' \rightarrow S.|\text{--}]$

   - enter A in ACTION[i, |--] (accept). NB - Conflict? (as in 2.)

ACTION table:

state	|--	,	a	b
0	X	X	S4	S5
1	A	S2	*	*
2	X	X	S4	S5
3	R1	R1	*	*
4	R3	R3	*	*
5	R4	R4	*	*
6	R2	R2	*	*

From DFA to parser tables: GOTO Table Summary

1. For each DFA transition $I_i \rightarrow I_j$ reading nonterminal A

   (i.e., I_i contains an item $[X \rightarrow \alpha.A\beta]$)

   - enter GOTO[i , A] = j

GOTO table:

state	L	E
0	1	6
1	*	*
2	*	3
3	*	*
4	*	*
5	*	*
6	*	*

TDDD55 Compilers and Interpreters
TDDB44 Compiler Construction

Conflicts and Categories of LR Grammars and Parsers

Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2011.

Conflict Examples in LR Grammars

- **Shift – Reduce conflict:**
  - E → id + E (shift +)
    | id (reduce id)
- **Reduce – Reduce conflict:**
  - E → id (reduce id)
    Pcall → id (reduce id)
- (Shift – Accept conflict)
  - S' → L (accept)
    L → L , E (shift ,)

Conflicts in LR Grammars

Observe conflicts in DFA (GOTO graph) kernels
or at the latest when filling the ACTION table.

- **Shift-Reduce conflict**
  - A DFA accepting state has an outgoing transition,
    i.e. contains items $[X \rightarrow \alpha.]$ and $[Y \rightarrow \beta.Z\gamma]$ for some Z in $N \cup \Sigma$
- **Reduce-Reduce conflict**
  - A DFA accepting state can reduce for multiple nonterminals
    i.e. contains at least 2 items $[X \rightarrow \alpha.]$ and $[Y \rightarrow \beta.]$, X != Y
- **(Shift/Reduce-Accept conflict)**
  - A DFA accepting state containing $[S' \rightarrow S.|\text{--}]$ contains
    another item $[X \rightarrow \alpha S.]$ or $[X \rightarrow \alpha S.b\beta]$

Only for LR(0) grammars there are no conflicts.

Handling Conflicts in LR Grammars

(Overview):

- Use lookahead
  - if lucky, the LR(0) states + a few fixed lookahead sets are sufficient to eliminate all conflicts in the LR(0)-DFA
    - SLR(1), LALR(1)
  - otherwise, use LR(1) items $[X \rightarrow \alpha.\beta, a]$ (a is look-ahead) to build new, larger NFA/DFA
    - expensive (many items/states → very large tables)
  - if still conflicts, may try again with *k*>1 → even larger tables
- Rewrite the grammar (factoring / expansion) and retry...
- If nothing helps, re-design your language syntax
  - Some grammars are not LR(*k*) for any constant *k*
    and cannot be made LR(*k*) by rewriting either

Look-Ahead (LA) Sets

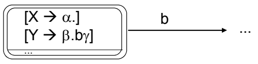
- For a LR(0) item $[X \rightarrow \alpha.\beta]$ in DFA-state I_i, define
  lookahead set $LA(I_i, [X \rightarrow \alpha.\beta])$ (a subset of Σ)

For SLR(1), LALR(1) etc., the LA sets only differ for reduce items:

- **For SLR(1):**
  $LA_{SLR}(I_i, [X \rightarrow \alpha.]) = \{ a \text{ in } \Sigma\text{: } S' \Rightarrow^* \beta X a \gamma \} = FOLLOW_1(X)$
  for all I_i with $[X \rightarrow \alpha.]$ in I_i
  - depends on nonterminal X only, not on state I_i
- **For LALR(1):**
  $LA_{LALR}(I_i, [X \rightarrow \alpha.]) = \{ a \text{ in } \Sigma\text{: } S' \Rightarrow^* \beta X a w$ and the LR(0)-DFA started in I_0 reaches I_i after reading $\beta\alpha \}$
  - usually a subset of $FOLLOW_1(X)$, i.e. of SLR LA set
  - depends on state I_i

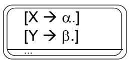
Made it simple:
Is my grammar SLR(1) ?

- Construct the (LR(0)-item) characteristic NFA
  and its equivalent DFA (= GOTO graph) as above.
- Consider all conflicts in the DFA states:
  - **Shift-Reduce:**



Consider all pairs of conflicting items $[X \rightarrow \alpha.]$, $[Y \rightarrow \beta.b\gamma]$:
If b in $FOLLOW_1(X)$ for any of these ➔ not SLR(1).

  - **Reduce-Reduce:**



Consider all pairs of conflicting items $[X \rightarrow \alpha.]$, $[Y \rightarrow \beta.]$:
If $FOLLOW_1(X)$ intersects with $FOLLOW_1(Y)$ ➔ not SLR(1)

  - (**Shift-Accept**: similar to Shift-Reduce)

Example: L-Values in C Language

- L-values on left hand side of assignment.
  Part of a C grammar:
  1. S' → S
  2. S → L = R
  3. | R
  4. L → *R
  5. | **id**
  6. R → L
- Avoids that R (for R-values) appears as LHS of assignments
- But *R = ... is ok.
- This grammar is LALR(1) but not SLR(1):

Example (cont.)

LR(0) parser has a shift-reduce conflict in kernel of state I_2:

- I_0 = { [S'→.S], [S→.L**=**R], [S→.R], [L→.*R], [L→.**id**], R→.L] }
- I_1 = { [S'->S.] }
- I_2 = { [S->L.**=**R], [R->L.] }
- I_3 = { [S->R.] }
- I_4 = { [L->*.R], [R->.L], [L->.*R], [L->.**id**] }
- I_5 = { [L->**id**.] }
- I_6 = { [S->L**=**.R], [R->.L], [L->.*R], L->.**id**] }
- I_7 = { [L->*R.] }
- I_8 = { [R->L.] }
- I_9 = { [S->L**=**R.] }

Shift **=** or reduce to R?

$FOLLOW_1(R)$ = { |– , **=** } ➔ SLR(1) still shift-reduce conflict in I_2
as **=** does not disambiguate

Example (cont.)

- I_0 = { [S'->.S], [S->.L**=**R], [S->.R], [L->.*R], [L->.**id**], R->.L] }
- I_1 = { [S'->S.] }
- I_2 = { [S->L.**=**R], [R->L.] }
- I_3 = { [S->R.] }
- I_4 = { [L->*.R], [R->.L], [L->.*R], [L->.**id**] }
- I_5 = { [L->**id**.] }
- I_6 = { [S->L**=**.R], [R->.L], [L->.*R], L->.**id**] }
- I_7 = { [L->*R.] }
- I_8 = { [R->L.] }
- I_9 = { [S->L**=**R.] }

LA_{LALR} (I_2, [R->L.]) = { |– } ➔ LALR(1) parser is conflict-free
as computation path $I_0 \ldots I_2$ does not really allow **=** following R.
= can only occur after R if ”*R” was encountered before.

LALR(1) Parser Construction

Method 1: (simple but not practical)

1. Construct the LR(1) items (see later). (If there is already a conflict, stop.)
2. Look for sets of LR(1) items that have the same kernel,
   and merge them.
3. Construct the ACTION table as for LR(1).
   If a conflict is detected, the grammar is not LALR(1).
4. Construct the GOTOgraph function:
   For each merged $J = I_1 \cup I_2 \cup \ldots \cup I_r$,
   the kernels of GOTOgr(I_1,X), ..., GOTOgr(I_r,X) are identical because the kernels of $I_1,\ldots,I_r$ are identical.
   Set GOTOgr(J, X) := $\bigcup$ { I: I has the same kernel as GOTOgr(I_1,X) }

Method 2: (practical, used) (details see textbook)

1. Start from LR(0) items and construct kernels of DFA states I_0, I_1, ...
2. Compute lookahead sets by propagation along the GOTOgr(I_j,X) edges
   (fixed point iteration).

Solve Conflicts by Rewriting the Grammar

- Eliminate Reduce-Reduce Conflict:

Factoring

S → (A) | (B)
A → char | integer | ident
B → float | double | ident

[A → ident .]
[B → ident .]
...

factor ident

S → (A) | (B) | (C)
A → char | integer
B → float | double
C → ident

- Eliminate Shift-Reduce Conflict: (one token lookahead: '(')

Inline-Expansion

S → (A) | OptY (B)
OptY → Y | ε
Y → ...
A → ...
B → ...

[S → . (A)]
[S → . OptY (B)]
[OptY → .Y]
[OptY → .ε]
[OptY → ε .]
[Y → ...] ...

expand OptY

S → (A) | (B)
| Y (B)
Y → ...
A → ...
B → ...

---

p.116

LR(*k*) Grammar - Formal Definition

- Let G' be the augmented grammar for G (i.e., extended by new start symbol S' and production rule S' –> S |--)
- G is called a **LR(*k*) grammar** if
  - $S' \; {}_{rm}{=}{>}^{*} \; \alpha X w \; {}_{rm}{=}{>} \; \alpha\beta w$ and
  - $S' \; {}_{rm}{=}{>}^{*} \; \gamma Y x \; {}_{rm}{=}{>} \; \alpha\beta y$ and
  - w[1:*k*] = y[1:*k*]

  imply that $\alpha = \gamma$ and X = Y and x = y = w.

i.e., considering at most *k* symbols after the handle, in each rightmost derivation the handle can be localized and the production to be applied can be determined.

Remark: w, x, y in Σ^* α, β, γ in $(N \cup \Sigma)^*$ X, Y in N

Some grammars are not LR(*k*) for any fixed *k*

- Example: S → a B c
  B → b B b
  | b
  - describes language { a b^{2N+1} c : N >= 0 }
- This grammar is not LR(*k*) for any fixed *k*.

Proof: As k is fixed (constant), consider for any n > *k*:

- S =>* a b^n B b^n c => a b^n $\underline{b}$ b^n c
- S =>* a b^{n+1} B b^{n+1} c => a b^{n+1} $\underline{b}$ b^{n+1} c

By the LR(*k*) definition,

- α = a b^n β = b w = b^n c
- γ = a b^{n+1} β = b y = b^{n+1} c

The handle cannot be localized with only limited lookahead size k

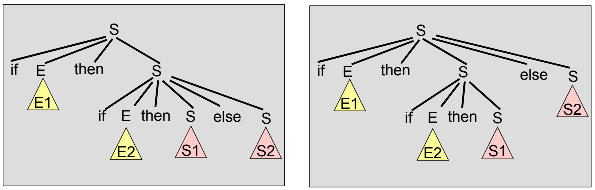
Although w[1:*k*] = y[1:*k*], we have α != γ ➔ grammar is not LR(*k*).

No ambiguous grammar is LR(*k*) for any fixed *k*

- S → **if** E **then** S
  | **if** E **then** S **else** S
  | *other statements*
  ...

is ambiguous – the following statement has 2 parse trees:

if E1 **then** **if** E2 **then** S1 **else** S2



(cont.)

- Consider situation
  (configuration of shift-reduce parser)

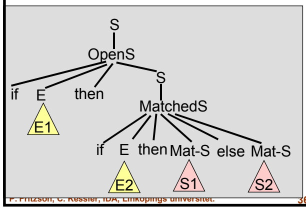
  --| ... **if** E **then** S **else** ... |--
- Not clear whether to
  - shift else
    (following production 2, i.e. **if** E **then** S is not handle), or
  - reduce handle **if** E **then** S to S (following production 1)
- Any fixed-size lookahead (else and beyond) does not help!
- Suggestion: Rewrite grammar to make it unambiguous

Rewriting the grammar...

S → MatchedS
| OpenS
MatchedS → **if** E **then** MatchedS **else** MatchedS
| *other statements*
OpenS → **if** E **then** S
| **if** E **then** MatchedS **else** OpenS
...

is no longer ambiguous

Impossible now to derive any sentential form containing an OpenS nonterminal from a MatchedS



Some grammars are not LR(*k*) for any fixed *k*

- Grammar with productions

  S → a S a | ε

  is unambiguous but not LR(*k*) for any fixed *k*. (Why?)

- An equivalent LR grammar for the same language is

  S → a a S | ε

LR(1) Items and LR(*k*) Items

LR(*k*) parser: Construction similar to LR(0) / SLR(1) parser, but plan for distinguishing between states for *k*>0 tokens **lookahead** already from the beginning

- States in the LR(0) GOTO graph may be split up

- LR(1) items:
  [A->$\alpha.\beta$, a] for all productions A->$\alpha\beta$ and all a in Σ
- Can be combined for lookahead symbols with equal behavior:
  [A->$\alpha.\beta$, a|b] or [A->$\alpha.\beta$, L] for a subset L of Σ
- Generalized to *k*>1:
  [A->$\alpha.\beta$, $a_1a_2...a_k$]

Interpretation of [A->$\alpha.\beta$, a] in a state:

- If β not ε, ignore second component (as in LR(0))
- If β=ε, *i.e.* [A->α. , a], reduce only if next input symbol = a.

LR(1) Parser

- NFA start state is [S'->.S, |–]
- Modify computation of *Closure*(*I*), GOTO(*I*,X) and the subset computation for LR(1) items
  - Details see [ASU86, p.232] or [ALSU06, p.261]
- Can have many more states than LR(0) parser
  - Which may help to resolve some conflicts

Interesting to know...

- For each LR(*k*) grammar with some constant *k*>1 there exists an equivalent* grammar G' that is LR(1).

- For any LL(*k*) grammar there exists an equivalent LR(*k*) grammar (but not vice versa!)
  - e.g., language { $a^n b^n$: n>0 } U { $a^n c^n$: n > 0 }
    has a LR(0) grammar
    but no LL(*k*) grammar for any constant *k*.

- Some grammars are LR(0) but not LL(*k*) for any *k*
  - e.g., S → A b
    A → Aa | a (left recursion, could be rewritten)

* Two grammars are *equivalent* if they describe the same language.