Compilation

0368-3133 (Semester A, 2013/14)

Lecture 6a: Syntax

(Bottom–up parsing)

Noam Rinetzky

Slides credit: Roman Manevich, Mooly Sagiv, Eran Yahav

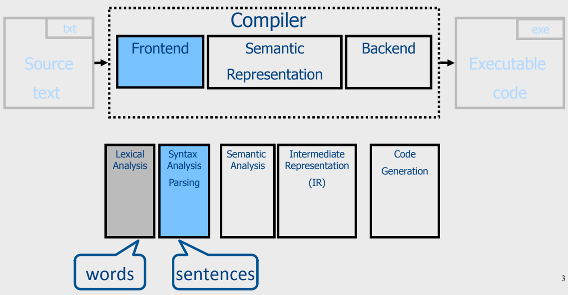
What is a Compiler?

"A compiler is a computer program that transforms source code written in a programming language (source language) into another language (target language).

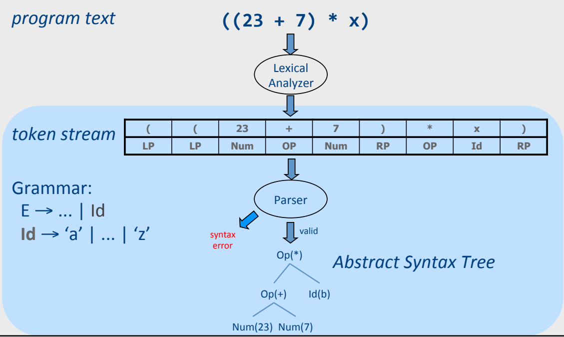
The most common reason for wanting to transform source code is to create an executable program."

--Wikipedia

Conceptual Structure of a Compiler



From scanning to parsing

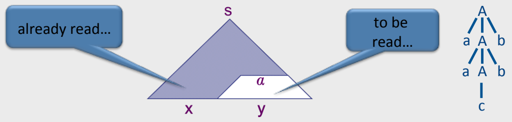


Top-Down vs Bottom-Up

A→aAb | c

aacbb$

- Top-down (predict match/scan-complete)

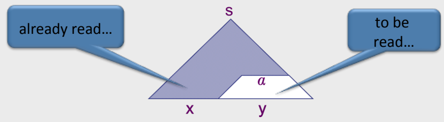


Top-Down vs Bottom-Up

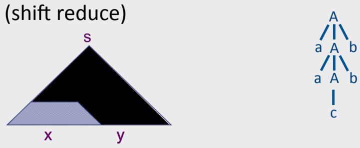
A→aAb | c

aacbb$

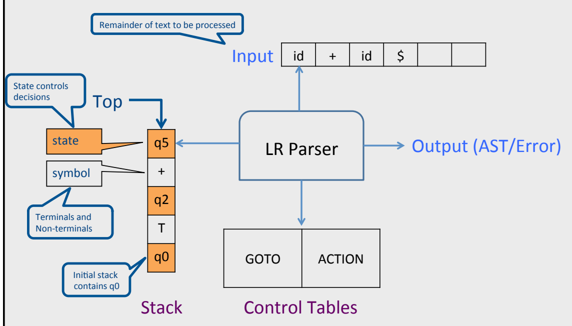
- Top-down (predict match/scan-complete)



- Bottom-up (shift reduce)



Model of an LR parser



LR(0) parser tables

Empty cell = error move

GOTO Table

ACTION Table

State	i	+	(	)	$	E	T	action
q0	q5		q7			q1	q6	shift
q1		q3			q2			shift
q2								reduce: Z→E$
q3	q5		q7				q4	shift
q4								reduce: E→E+T
q5								reduce: T→i
q6								reduce: E→T
q7	q5		q7			q8	q6	shift
q8		q3		q9				shift
q9								reduce: T→E

LR(0) parser tables

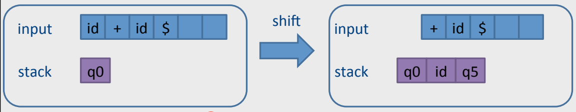
- Shift action row
  - Tells which state to GOTO for current token
  - Blank entry indicates an error
- Reduce action row
  - Tells which rule to reduce with
    - Independent of current token
  - GOTO entries are blank

State	id	+	(	)	$	E	T	action
q0	q5		q7			q1	q6	shift
q5								reduce: T→i

9

Shift Move

- Remove first token from input
- Push it on the stack
- Compute next state based on GOTO table
- Push new state on the stack
- If new state is error – report error

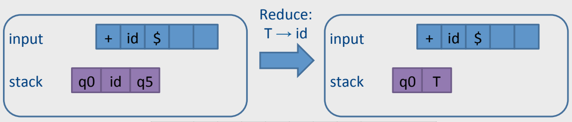


State	id	+	(	)	$	E	T	action
q0	q5		q7			q1	q6	shift
q5								reduce: T→i

10

Reduce Move (using N→α)

- Symbols in α and their following states are removed from stack
- New state computed based on GOTO table (using top of stack, before pushing N)
- N is pushed on the stack
- New state pushed on top of N

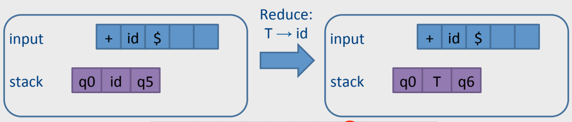


State	id	+	(	)	$	E	T	action
q0	q5		q7			q1	q6	shift
q5								reduce: T→i

11

Reduce Move (using N→α)

- Symbols in α and their following states are removed from stack
- New state computed based on GOTO table (using top of stack, before pushing N)
- N is pushed on the stack
- New state pushed on top of N



State	id	+	(	)	$	E	T	action
q0	q5		q7			q1	q6	shift
q5								reduce: T→i

12

GOTO/ACTION table

State	i	+	(	)	$	E	T
q0	s5		s7			s1	s6
q1		s3			s2		
q2	r1	r1	r1	r1	r1	r1	r1
q3	s5		s7				s4
q4	r3	r3	r3	r3	r3	r3	r3
q5	r4	r4	r4	r4	r4	r4	r4
q6	r2	r2	r2	r2	r2	r2	r2
q7	s5		s7			s8	s6
q8		s3		s9			
q9	r5	r5	r5	r5	r5	r5	r5

(1) Z → E $
(2) E → T
(3) E → E + T
(4) T → i
(5) T → (E)

Warning: numbers mean different things!
rn = reduce using rule number n
sm = shift to state m

13

Parsing id+id$

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Stack	Input	Action
0	id + id $	s5

Initialize with state 0

S	action					goto	
	id	+	(	)	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

14

Parsing id+id$

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Stack	Input	Action
0	id + id $	s5

Initialize with state 0

S	action					goto	
	id	+	(	)	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

15

Parsing id+id$

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Stack	Input	Action
0	id + id $	s5
0 id 5	+ id $	r4

S	action					goto	
	id	+	(	)	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

16

Parsing id+id$

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Stack	Input	Action
0	id + id $	s5
0 id 5	+ id $	r4

pop id 5

S	*action*					*goto*	
	id	+	(	)	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

Parsing id+id$

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Stack	Input	Action
0	id + id $	s5
0 ~~id 5~~	+ id $	r4

push T 6

S	*action*					*goto*	
	id	+	(	)	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

Parsing id+id$

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Stack	Input	Action
0	id + id $	s5
0 id 5	+ id $	r4
0 T 6	+ id $	r2

S	*action*					*goto*	
	id	+	(	)	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

Parsing id+id$

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Stack	Input	Action
0	id + id $	s5
0 id 5	+ id $	r4
0 T 6	+ id $	r2
0 E 1	+ id $	s3

S	*action*					*goto*	
	id	+	(	)	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

Parsing id+id$

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Stack	Input	Action
0	id + id $	s5
0 id 5	+ id $	r4
0 T 6	+ id $	r2
0 E 1	+ id $	s3
0 E 1 + 3	id $	s5

S	action					goto	
	id	+	(	)	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

21

Parsing id+id$

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Stack	Input	Action
0	id + id $	s5
0 id 5	+ id $	r4
0 T 6	+ id $	r2
0 E 1	+ id $	s3
0 E 1 + 3	id $	s5
0 E 1 + 3 id 5	$	r4

S	action					goto	
	id	+	(	)	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

22

Parsing id+id$

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Stack	Input	Action
0	id + id $	s5
0 id 5	+ id $	r4
0 T 6	+ id $	r2
0 E 1	+ id $	s3
0 E 1 + 3	id $	s5
0 E 1 + 3 id 5	$	r4
0 E 1 + 3 T 4	$	r3

S	action					goto	
	id	+	(	)	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

23

Parsing id+id$

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

Stack	Input	Action
0	id + id $	s5
0 id 5	+ id $	r4
0 T 6	+ id $	r2
0 E 1	+ id $	s3
0 E 1 + 3	id $	s5
0 E 1 + 3 id 5	$	r4
0 E 1 + 3 T 4	$	r3
0 E 1	$	s2

S	action					goto	
	id	+	(	)	$	E	T
0	s5		s7			g1	g6
1		s3			acc		
2							
3	s5		s7				g4
4	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4		
6	r2	r2	r2	r2	r2		
7	s5		s7			g8	g6
8		s3		s9			
9	r5	r5	r5	r5	r5		

24

Constructing an LR parsing table

- Construct a transition diagram (deterministic FSM)
  - States = sets of LR(0) items
  - Transitions = one-step derivation
- If there are conflicts – stop
- Fill table entries from diagram

25

Terminology: Reductions & Handles

- The opposite of derivation is called *reduction*
  - Let A→α be a production rule
  - Derivation: βAµ ➔ βαµ
  - Reduction: βαµ ➔ βAµ
- A *handle* is the reduced substring
  - α is the handles for βαµ

26

LR(0) Items

- The items of a grammar are obtained by placing a dot at every position in every production

Grammar

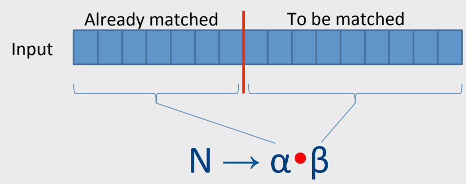
(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

LR(0) items

1: S → •E$
2: S → E • $
3: S → E $ •
4: E → • T
5: E → T •
6: E → • E + T
7: E → E • + T
8: E → E + • T
9: E → E + T •
10: T → • id
11: T → id •
12: T → • (E)
13: T → (• E)
14: T → (E •)
15: T → (E) •

27

LR(0) Item - Intuition



Hypothesis about αβ is the rule being reduced, and so far we've matched α and we expect to see β

28

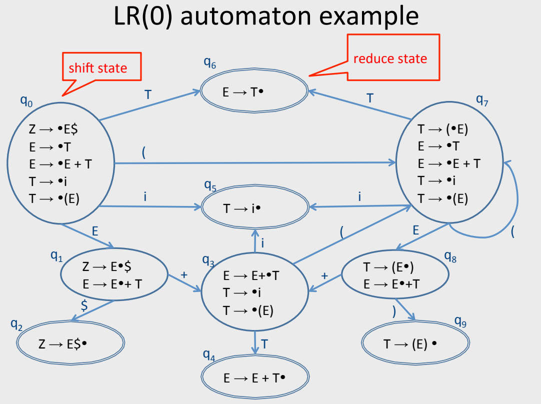
Types of LR(0) items

$N \longrightarrow \alpha \bullet \beta$ Shift Item

$N \longrightarrow \alpha\beta \bullet$ Reduce Item

29

LR(0) automaton example



30

Computing item sets

- Initial set
  - Z is in the start symbol
  - ε-closure({ Z⟶•α | Z⟶α is in the grammar })
- Next set from a set S and the next symbol X
  - step(S,X) = { N→αX•β | N→α•Xβ in the item set S}
  - nextSet(S,X) = ε-closure(step(S,X))

31

Operations for transition diagram construction

- Initial = {S'⟶•S$}
- For an item set I
  Closure(I) = Closure(I) ∪
  {X⟶•μ is in grammar| N⟶α•Xβ in I}
- Goto(I, X) = { N⟶αX•β | N⟶α•Xβ in I}

32

Initial example

Grammar

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

- Initial = {S → •E $}

Closure example

Grammar

(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

- Initial = {S → •E $}
- Closure({S → •E $}) = {
  S → •E $
  E → •T
  E → •E + T
  T → •id
  T → •(E) }

Goto example

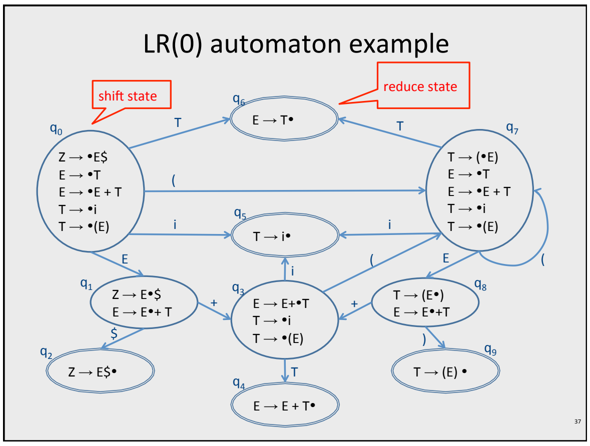
Grammar

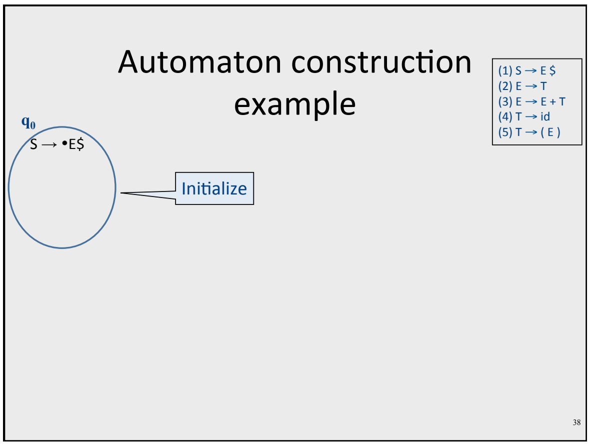
(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)

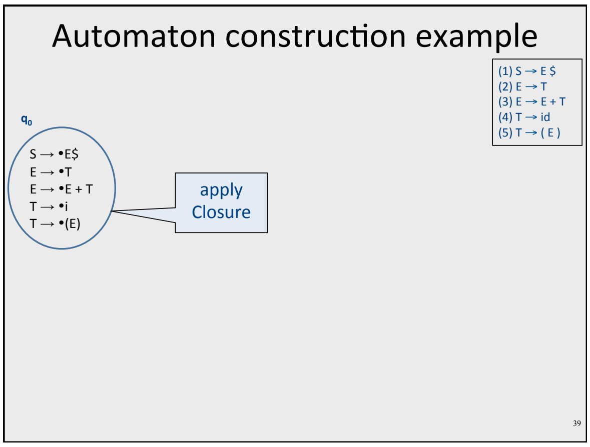
- Initial = {S → •E $}
- Closure({S → •E $}) = {
  S → •E $
  E → •T
  E → •E + T
  T → •id
  T → •(E) }
- Goto({S → •E $, E → •E + T, T → •id}, E) =
  {S → E• $, E → E• + T}

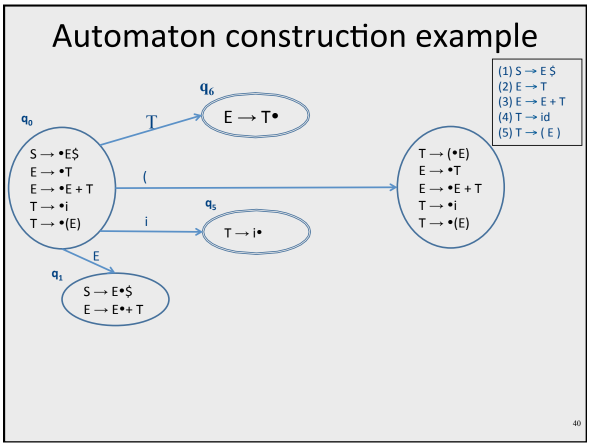
Constructing the transition diagram

- Start with state 0 containing item Closure({S → •E $})
- Repeat until no new states are discovered
  - For every state p containing item set Ip, and symbol N, compute state q containing item set Iq = Closure(goto(Ip, N))

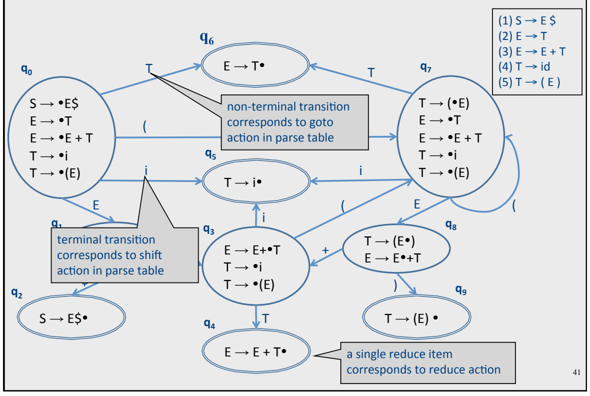

LR(0) automaton example
shift state
reduce state
q0
Z → •E$
E → •T
E → •E + T
T → •i
T → •(E)
q6
E → T•
q7
T → (•E)
E → •T
E → •E + T
T → •i
T → •(E)
q5
T → i•
q1
Z → E•$
E → E•+ T
q3
E → E+•T
T → •i
T → •(E)
q8
T → (E•)
E → E•+T
q2
Z → E$•
q4
E → E + T•
q9
T → (E) •
37


Automaton construction example
(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)
q0
S → •E$
Initialize
38


Automaton construction example
(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)
q0
S → •E$
E → •T
E → •E + T
T → •i
T → •(E)
apply Closure
39


Automaton construction example
(1) S → E $
(2) E → T
(3) E → E + T
(4) T → id
(5) T → (E)
q0
S → •E$
E → •T
E → •E + T
T → •i
T → •(E)
q6
E → T•
T → (•E)
E → •T
E → •E + T
T → •i
T → •(E)
q5
T → i•
q1
S → E•$
E → E•+ T
40

Automaton construction example



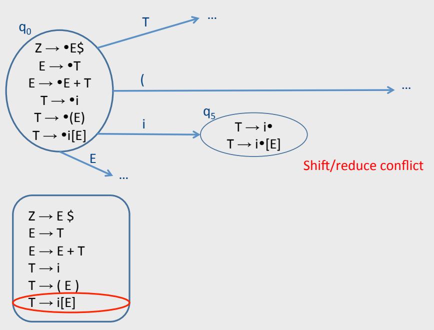
41

Are we done?

- Can make a transition diagram for any grammar
- Can make a GOTO table for every grammar
- Cannot make a deterministic ACTION table for every grammar

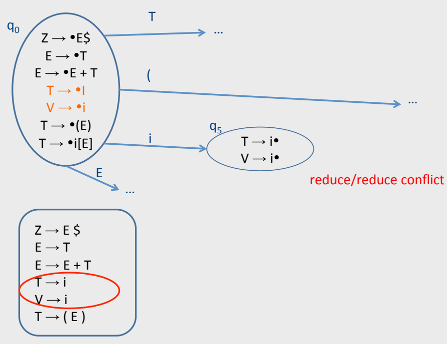
42

LR(0) conflicts



43

LR(0) conflicts



44

LR(0) conflicts

- Any grammar with an ε-rule cannot be LR(0)
- Inherent shift/reduce conflict
  - A→ ε• – reduce item
  - P →α•Aβ – shift item
  - A→ ε• can always be predicted from P →α•Aβ

Conflicts

- Can construct a diagram for every grammar but some may introduce conflicts
- shift-reduce conflict: an item set contains at least one shift item and one reduce item
- reduce-reduce conflict: an item set contains two reduce items

LR variants

- LR(0) – what we've seen so far
- SLR(0)
  - Removes infeasible reduce actions via FOLLOW set reasoning
- LR(1)
  - LR(0) with one lookahead token in items
- LALR(0)
  - LR(1) with merging of states with same LR(0) component

LR (0) GOTO/ACTIONS tables

GOTO table is indexed by state and a grammar symbol from the stack

GOTO Table

ACTION Table

State	i	+	(	)	$	E	T	action
q0	q5		q7			q1	q6	shift
q1		q3			q2			shift
q2								Z→E$
q3	q5		q7				q4	Shift
q4								E→E+T
q5								T→i
q6								E→T
q7	q5		q7			q8	q6	shift
q8		q3		q9				shift
q9								T→E

ACTION table determined only by state, ignores input

SLR parsing

- A handle should not be reduced to a non-terminal N if the lookahead is a token that cannot follow N
- A reduce item N → α• is applicable only when the lookahead is in FOLLOW(N)
  - If b is not in FOLLOW(N) we just proved there is no derivation S ➔* βNb.
  - Thus, it is safe to remove the reduce item from the conflicted state
- Differs from LR(0) only on the ACTION table
  - Now a row in the parsing table may contain both shift actions and reduce actions and we need to consult the current token to decide which one to take

49

SLR action table

Lookahead token from the input

State	i	+	(	)	[	]	$
0	shift		shift				
1		shift					accept
2							
3	shift		shift				
4		E→E+T		E→E+T			E→E+T
5		T→i		T→i	shift		T→i
6		E→T		E→T			E→T
7	shift		shift				
8		shift		shift			
9		T→(E)		T→(E)			T→(E)

SLR – use 1 token look-ahead

vs.

state	action
q0	shift
q1	shift
q2	
q3	shift
q4	E→E+T
q5	T→i
q6	E→T
q7	shift
q8	shift
q9	T→E

LR(0) – no look-ahead

```
… as before…
T → i
T → i[E]
```

50

LR(1) grammars

- In SLR: a reduce item N → α• is applicable only when the lookahead is in FOLLOW(N)
- But FOLLOW(N) merges lookahead for all alternatives for N
  - Insensitive to the context of a given production
- LR(1) keeps lookahead with each LR item
- Idea: a more refined notion of follows computed per item

51

LR(1) items

- LR(1) item is a pair
  - LR(0) item
  - Lookahead token
- Meaning
  - We matched the part left of the dot, looking to match the part on the right of the dot, followed by the lookahead token
- Example
  - The production L → id yields the following LR(1) items

(0) S' → S
(1) S → L = R
(2) S → R
(3) L → * R
(4) L → id
(5) R → L

LR(0) items

[L → ● id]
[L → id ●]

LR(1) items

[L → ● id, *]
[L → ● id, =]
[L → ● id, id]
[L → ● id, $]
[L → id ●, *]
[L → id ●, =]
[L → id ●, id]
[L → id ●, $]

52

LALR(1)

- LR(1) tables have huge number of entries
- Often don't need such refined observation (and cost)
- Idea: find states with the same LR(0) component and merge their lookaheads component as long as there are no conflicts
- LALR(1) not as powerful as LR(1) in theory but works quite well in practice
  - Merging may not introduce new shift-reduce conflicts, only reduce-reduce, which is unlikely in practice

53

Summary

54

Summary

- Bottom up derivation
- LR(k) can decide on a reduce after seeing the entire right side of the rule plus k look-ahead tokens.
  - We focused on LR(0)
- Using a table and a stack to derive
- LR Items and the automaton
- Creating the table from the automaton
- LR parsing with pushdown automata
- LR(0), SLR, LR(1) – different kinds of LR items, same basic algorithm

55

Broad kinds of parsers

- Parsers for arbitrary grammars
  - Earley's method, CYK method
  - Usually, not used in practice (though might change)
- Top-Down parsers
  - Construct parse tree in a top-down matter
  - Find the leftmost derivation
- Bottom-Up parsers
  - Construct parse tree in a bottom-up manner
  - Find the rightmost derivation in a reverse order

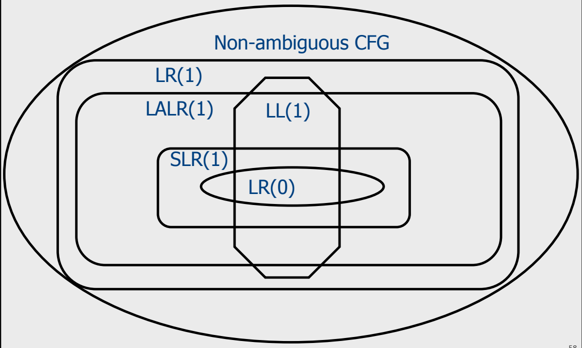
56

LR is More Powerful than LL

- Any LL(k) language is also in LR(k), i.e., LL(k) ⊂ LR(k).
  - LR is more popular in automatic tools
- But less intuitive
- Also, the lookahead is counted differently in the two cases
  - In an LL(k) derivation the algorithm sees the left-hand side of the rule + k input tokens and then must select the derivation rule
  - In LR(k), the algorithm "sees" all right-hand side of the derivation rule + k input tokens and then reduces
    - LR(0) sees the entire right-side, but no input token

57

Grammar Hierarchy



58