Parsing

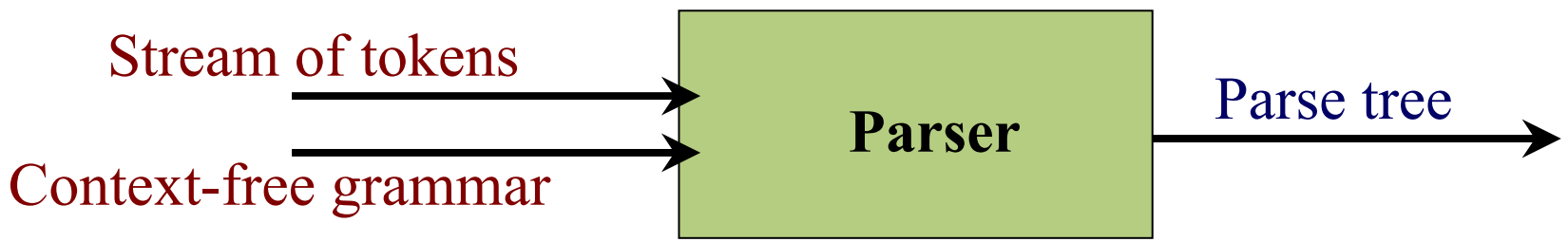
Jaruloj Chongstitvatana
Department of Mathematics and Computer Science
Chulalongkorn University

Outline

- Top-down parsing
  - Recursive-descent parsing
  - LL(1) parsing
    - LL(1) parsing algorithm
    - First and follow sets
    - Constructing LL(1) parsing table
    - Error recovery
- Bottom-up parsing
  - Shift-reduce parsers
  - LR(0) parsing
    - LR(0) items
    - Finite automata of items
    - LR(0) parsing algorithm
    - LR(0) grammar
  - SLR(1) parsing
    - SLR(1) parsing algorithm
    - SLR(1) grammar
    - Parsing conflict

Introduction

- Parsing is a process that constructs a syntactic structure (i.e. parse tree) from the stream of tokens.
- We already learn how to describe the syntactic structure of a language using (context-free) grammar.
- So, a parser only need to do this?



Top–Down Parsing

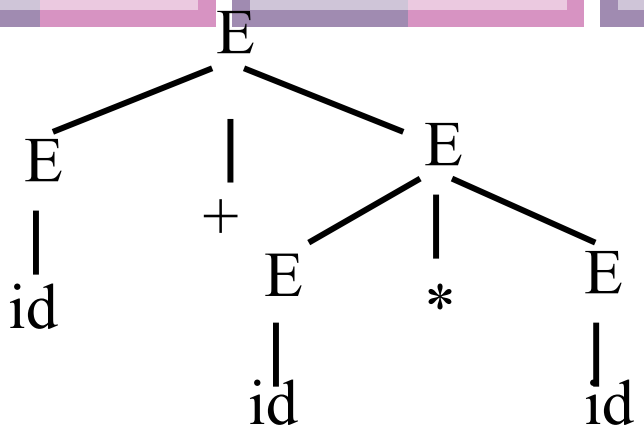
- A parse tree is created from root to leaves
- The traversal of parse trees is a preorder traversal
- Tracing leftmost derivation
- Two types:
  - Backtracking parser
  - Predictive parser

Guess the structure of the parse tree from the next input

Bottom–Up Parsing

- A parse tree is created from leaves to root
- The traversal of parse trees is a reversal of postorder traversal
- Tracing rightmost derivation
- Try different structures and backtrack if it does not matched the input

Parse Trees and Derivations



Top-down parsing

E ⇒ E + E

⇒ id + E

⇒ id + E * E

⇒ id + id * E

⇒ id + id * id

ไม่ใช่แบบ bottom up parsing

Top-down Parsing

- What does a parser need to decide?
  - Which production rule is to be used at each point of time ?
- How to guess?
- What is the guess based on?
  - What is the next token?
    - Reserved word if, open parentheses, etc.
  - What is the structure to be built?
    - If statement, expression, etc.

Top-down Parsing

- Why is it difficult?
  - Cannot decide until later
    - Next token: **if** Structure to be built: St
    - St ⇒ MatchedSt | UnmatchedSt
    - UnmatchedSt ⇒
      if (E) St| **if** (E) MatchedSt **else** UnmatchedSt
    - MatchedSt ⇒ **if** (E) MatchedSt **else** MatchedSt |...
  - Production with empty string
    - Next token: **id** Structure to be built: par
    - par ⇒ parList | ε
    - parList ⇒ exp , parList | exp

Recursive-Descent

- Write one procedure for each set of productions with the same nonterminal in the LHS
- Each procedure recognizes a structure described by a nonterminal.
- A procedure calls other procedures if it need to recognize other structures.
- A procedure calls *match* procedure if it need to recognize a terminal.

Recursive-Descent: Example

E → E O F | F
O → + | -
F → (E) | id

E ::= F {O F}
O ::= + | -
F ::= (E) | id

```
procedure F
{    switch token
   {    case (:  match('(');
                 E;
                 match(')');
        case id: match(id);
        default:  error;
   }
}
```

```
procedure E
{    E; O; F; }
```

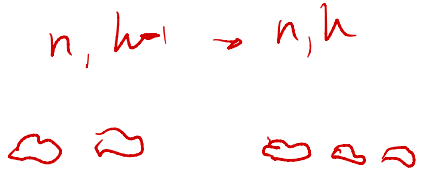
- For this grammar:
  - We cannot decide which rule to use for E, and
  - If we choose E → E O F, it leads to infinitely recursive loops.
- Rewrite the grammar into EBNF

```
procedure E
{    F;
   while (token=+ or token=-)
   {    O; F;    }
}
```

Match procedure

```
procedure match(expTok)
{          if (token==expTok)
         then     getToken
         else     error
}
```

- The token is not consumed until **`getToken`** is executed.



Problems in Recursive-Descent

- Difficult to convert grammars into EBNF
- Cannot decide which production to use at each point
- Cannot decide when to use □-production A□ □

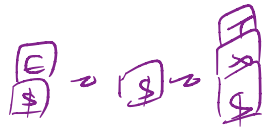
LL(1) Parsing

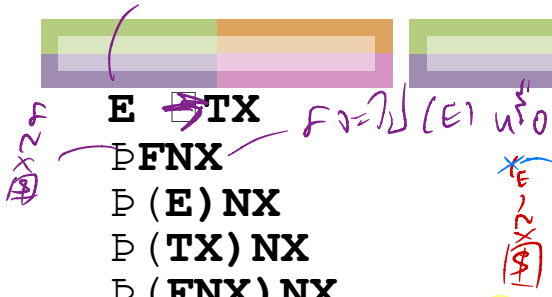
- LL(1)
  - Read input from (**L**) left to right
  - Simulate (**L**) leftmost derivation
  - **1** lookahead symbol
- Use stack to simulate leftmost derivation
  - Part of sentential form produced in the leftmost derivation is stored in the stack.
  - Top of stack is the leftmost nonterminal symbol in the fragment of sentential form.

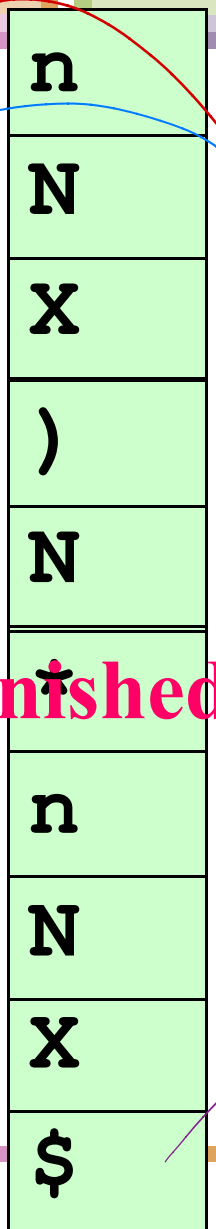
Concept of LL(1) Parsing

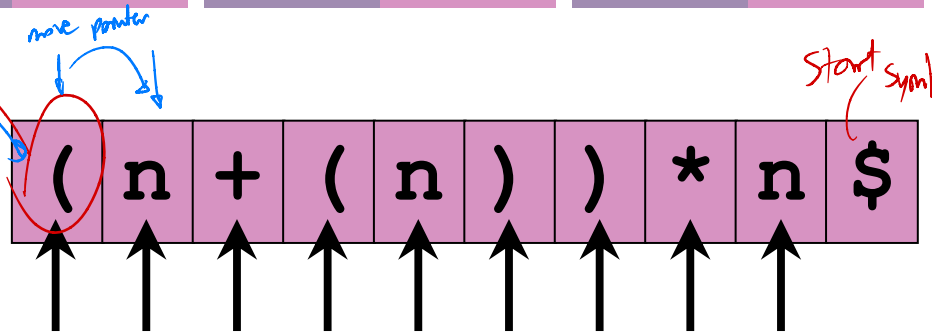
- Simulate leftmost derivation of the input.
- Keep part of sentential form in the stack.
- If the symbol on the top of stack is a terminal, try to match it with the next input token and pop it out of stack.
- If the symbol on the top of stack is a nonterminal X, replace it with Y if we have a production rule X □ Y.
  - Which production will be chosen, if there are both X □ Y and X □ Z ?

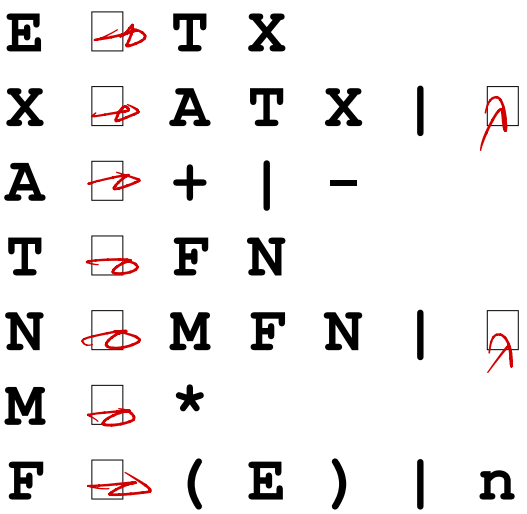
Example of LL(1) Parsing

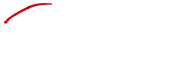


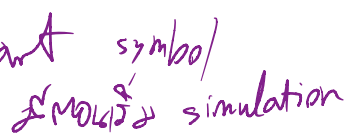












LL(1) Parsing Algorithm

Push the start symbol into the stack

WHILE stack is not empty ($ is not on top of stack) and the stream of tokens is not empty (the next input token is not $)

SWITCH (Top of stack, next token)

CASE (terminal a, a):

Pop stack; Get next token

CASE (nonterminal A, terminal a):

IF the parsing table entry M[A, a] is not empty THEN

Get A→X1 X2 ... Xn from the parsing table entry M[A, a] Pop stack;

Push Xn ... X2 X1 into stack in that order

ELSE Error — ไวยากรณ์ rule ที่ match

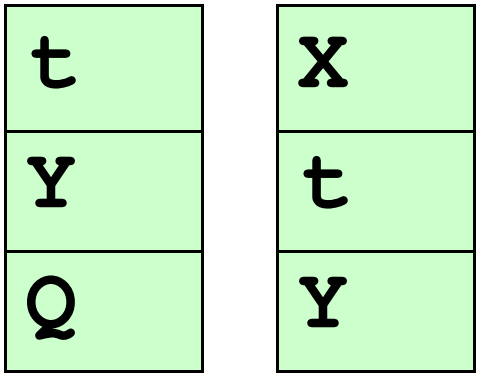
CASE ($,$): Accept

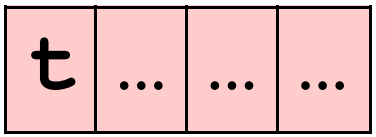
OTHER: Error

LL(1) Parsing Table

If the nonterminal *N* is on the top of stack and the next token is *t*, which production rule to use?

- Choose a rule $N \rightarrow X$ such that
  - $X \rightarrow^* tY$ or
  - $X \rightarrow^* \varepsilon$ and $S \rightarrow^* WNtY$





First Set

- Let X be λ or be in V or T.
- First(X) is the set of the first terminal in any sentential form derived from X.
  - If X is a terminal or λ, then First(X) = {X}.
  - If X is a nonterminal and $X \rightarrow X1\ X2 \ldots Xn$ is a rule, then
    - First($X1$) - {λ} is a subset of First(X)
    - First(Xi) - {λ} is a subset of First(X) if for all $j<i$ First(Xj) contains {λ}
    - λ is in First(X) if for all $j \leq n$ First(Xj) contains λ

Examples of First Set

exp → exp addop term |
term

addop → + | -

term → term mulop factor |
factor

mulop → *

factor → (exp) | num

First(addop) = {+, -}

First(mulop) = {*}

First(factor) = {(, num}

First(term) = {(, num}

First(exp) = {(, num}

st → ifst | other

ifst → if (exp) st elsepart

elsepart → else st | ε

exp → 0 | 1

First(exp) = {0,1}

First(elsepart) = {else, ε}

First(ifst) = {if}

First(st) = {if, other}

Algorithm for finding First(A)

For all terminals a, First(a) = {a}

For all nonterminals A, First(A) := { }

While there are changes to any First(A)

For each rule A → X1 X2 ... Xn

For each Xi in {X1, X2, …, Xn }

If for all j<i First(Xj) contains ε,

Then

add First(Xi)-{ε} to First(A)

If ε is in First(X1), First(X2), ..., and First(Xn)

Then add ε to First(A)

If A is a terminal or ε, then First(A) = {A}.

If A is a nonterminal, then for each rule A → X1 X2 ... Xn, First(A) contains First(X1) - {ε}.

If also for some i<n, First(X1), First(X2), ..., and First(Xi) contain ε, then First(A) contains First(Xi+1)-{ε}.

If First(X1), First(X2), ..., and First(Xn) contain ε, then First(A) also contains ε

Finding First Set: An Example

exp → term exp'

exp' → addop term exp' | ε

addop → + | -

term → factor term'

term' → mulop factor term' | ε

mulop → *

factor → (exp) | num

	First
exp	(num
exp'	+, -, ε
addop	+ -
term	(num
term'	*, ε
mulop	*
factor	(num

Follow Set

- Let $ denote the end of input tokens
- If A is the start symbol, then $ is in Follow(A).
- If there is a rule B ⇒ X A Y, then First(Y) - {λ} is in Follow(A).
- If there is production B ⇒ X A Y and λ is in First(Y), then Follow(A) contains Follow(B).

Algorithm for Finding Follow(A)

Follow(S) = {$}

FOR each A in V-{S}

Follow(A)={}

WHILE change is made to some Follow sets

FOR each production A □ X1 X2 ... Xn,

FOR each nonterminal Xi

Add First(Xi+1 Xi+2...Xn)-{□} into Follow(Xi).

(NOTE: If i=n, Xi+1 Xi+2...Xn= □)

IF □ is in First(Xi+1 Xi+2...Xn) THEN

Add Follow(A) to Follow(Xi)

If A is the start symbol, then $ is in Follow(A).

If there is a rule A □ Y X Z, then First(Z) - {□} is in Follow(X).

If there is production B □ X A Y and □ is in First(Y), then Follow(A) contains Follow(B).

Finding Follow Set: An Example

exp → term exp'

exp' → addop term exp' | ε

addop → + | -

term → factor term'

term' → mulop factor term' | ε

mulop → *

factor → (exp) | num

	First	Follow
exp	(num	$)
exp'	ε + -	$)
addop	+ -	(num
term	(num	+ - $)
term'	ε *	+ - $)
mulop	*	(num
factor	(num	* + - $)

Constructing LL(1) Parsing Tables

FOR each nonterminal A and a production A → X

FOR each token a in First(X)

A → X is in M(A, a)

IF → is in First(X) THEN

FOR each element a in Follow(A)

Add A → X to M(A, a)

Example: Constructing LL(1) Parsing Table

	First	Follow
exp	{(, num}	{$,)}
exp'	{+,-, □}	{$,)}
addop	{+,-}	{(,num}
term	{(,num}	{+,-,),$}
term'	{*, □}	{+,-,),$}
mulop	{*}	{(,num}
factor	{(, num}	{*,+,-,),$}

1 exp □ term exp'
2 exp' □ addop term exp'
3 exp' □ □
4 addop □ +
5 addop □ -
6 term □ factor term'
7 term' □ mulop factor term'
8 term' □ □
9 mulop □ *
10 factor □ (exp)
11 factor □ num

	(	)	+	-	*	num	$
exp	1					1	
exp'		3	2	2			3
addop			4	5			
term	6					6	
term'		8	8	8	7		8
mulop					9		
factor	10					11	

LL(1) Grammar

- A grammar is an LL(1) grammar if its LL(1) parsing table has at most one production in each table entry.

LL(1) Parsing Table for non-LL(1) Grammar

1 exp → exp addop term
2 exp → term
3 term → term mulop factor
4 term → factor
5 factor → (exp)
6 factor → num
7 addop → +
8 addop → -
9 mulop → *

	(	)	+	-	*	num	$
exp	1,2					1,2	
term	3,4					3,4	
factor	5					6	
addop			7	8			
mulop					9		

First(exp) = { (, num }
First(term) = { (, num }
First(factor) = { (, num }
First(addop) = { +, - }
First(mulop) = { * }

Causes of Non-LL(1) Grammar

- What causes grammar being non-LL(1)?
  - Left-recursion
  - Left factor

Left Recursion

- Immediate left recursion
  - A => A X | Y
  - A => A X1 | A X2 |...| A Xn | Y1 | Y2 |... | Ym
- Can be removed very easily
  - A => Y A', A' => X A' | ε
  - A => Y1 A' | Y2 A' |...| Ym A'
  - A' => X1 A' | X2 A' |…| Xn A' | ε
- General left recursion
  - A={Y1, Y2, …, Ym}{X1, X2, …, Xn}* A => X =>* A Y
- Can be removed when there is no empty-string production and no cycle in the grammar

Removal of Immediate Left Recursion

exp → exp + term | exp - term | term

term → term * factor | factor

factor → (exp) | num

- Remove left recursion

exp → term exp’ exp = term (+ term)*

exp’ → + term exp’ | - term exp’ | ε

term → factor term’ term = factor (* factor)*

term’ → * factor term’ | ε

factor → (exp) | num

General Left Recursion

- Bad News!
  - Can only be removed when there is no empty-string production and no cycle in the grammar.
- Good News!!!!
  - Never seen in grammars of any programming languages

君 君 私 私 君

てめえ

きさま

お前

Left Factoring

- Left factor causes non-LL(1)
  - Given A → X Y | X Z. Both A → X Y and A → X Z can be chosen when A is on top of stack and a token in First(X) is the next token.

A → X Y | X Z

can be left-factored as

A → X A' and A' → Y | Z

Example of Left Factor

ifSt → **if (** exp **)** st **else** st | **if (** exp **)** st

can be left-factored as

ifSt → **if (** exp **)** st elsePart

elsePart → **else** st | ε

seq → st **;** seq | st

can be left-factored as

seq → st seq'

seq' → **;** seq | ε

Outline

- Top-down parsing
  - Recursive-descent parsing
  - LL(1) parsing
    - LL(1) parsing algorithm
    - First and follow sets
    - Constructing LL(1) parsing table
    - Error recovery
- Bottom-up parsing
  - Shift-reduce parsers
  - LR(0) parsing
    - LR(0) items
    - Finite automata of items
    - LR(0) parsing algorithm
    - LR(0) grammar
  - SLR(1) parsing
    - SLR(1) parsing algorithm
    - SLR(1) grammar
    - Parsing conflict

Bottom-up Parsing

- Use explicit stack to perform a parse
- Simulate rightmost derivation (R) from left (L) to right, thus called LR parsing
- More powerful than top-down parsing
  - Left recursion does not cause problem
- Two actions
  - Shift: take next input token into the stack
  - Reduce: replace a string B on top of stack by a nonterminal A, given a production A □ B

Example of Shift-reduce Parsing

- Grammar
  S' □ S
  S □ (S)S | □
- Parsing actions

Stack	Input	Action
$	(( )) $	shift
$ (	( )) $	shift
$ ((	)) $	reduce
$ ((S	)) $	shift
$ ((S)	) $	reduce
$ ((S) S	) $	reduce S
$ (S	) $	shift
$ (S)	$	reduce S
$ (S) S	$	reduce S
$ S	$	accept

- Reverse of
- rightmost derivation
- from left to right

1 □ (( ))
2 □ (( ))
3 □ (( ))
4 □ ((S))
5 □ ((S))
6 □ ((S) S)
7 □ (S)
8 □ (S)
9 □ (S) S
10 S' □ S

Example of Shift-reduce Parsing

- Grammar
  S' □ S
  S □ (S)S | □
- Parsing actions

Stack	Input	Action
$	(( )) $	shift
$ (	( )) $	shift
$ ((	)) $	reduce S □ □
$ ((S	)) $	shift
$ ((S)	) $	reduce S □ □
$ ((S) S	) $	reduce S □ (S) S
$ (S	) $	shift
$ (S)	$	reduce S □ □
$ (S) S	$	reduce S □ (S) S
$ S	$	accept

Viable prefix

1	□ (( ))	
2	□ (( ))	
3	□ (( ))	
4	□ ((S))	
5	□ ((S))	
6	□ ((S) S)	
7	□ (S)	
8	□ (S)	
9	□ (S) S	
10	S' □ S	

handle

Terminologies

- Right sentential form
  - sentential form in a rightmost derivation
- Viable prefix
  - sequence of symbols on the parsing stack
- Handle
  - right sentential form + position where reduction can be performed + production used for reduction
- LR(0) item
  - production with distinguished position in its RHS

- Right sentential form
  - (S) S
  - ((S) S)
- Viable prefix
  - (S) S, (S), (S, (
  - ((S) S, ((S), ((S , ((, (
- Handle
  - (S) S. with S → ε
  - (S) S . with S → ε
  - ((S) S .) with S → (S) S
- LR(0) item
  - S → (S) S.
  - S → (S) . S
  - S → (S .) S
  - S → (. S) S
  - S → . (S) S

Shift-reduce parsers

- There are two possible actions:
  - shift and reduce
- Parsing is completed when
  - the input stream is empty and
  - the stack contains only the start symbol
- The grammar must be *augmented*
  - a new start symbol S' is added
  - a production S' → S is added
    - To make sure that parsing is finished when S' is on top of stack because S' never appears on the RHS of any production.

LR(0) parsing

- Keep track of what is left to be done in the parsing process by using finite automata of items
  - An item A → w . B y means:
    - A → w B y might be used for the reduction in the future,
    - at the time, we know we already construct w in the parsing process,
    - if B is constructed next, we get the new item A → w B . Y

LR(0) items

- LR(0) item
  - production with a distinguished position in the RHS
- Initial Item
  - Item with the distinguished position on the leftmost of the production
- Complete Item
  - Item with the distinguished position on the rightmost of the production
- Closure Item of x
  - Item x together with items which can be reached from x via □-transition
- Kernel Item
  - Original item, not including closure items

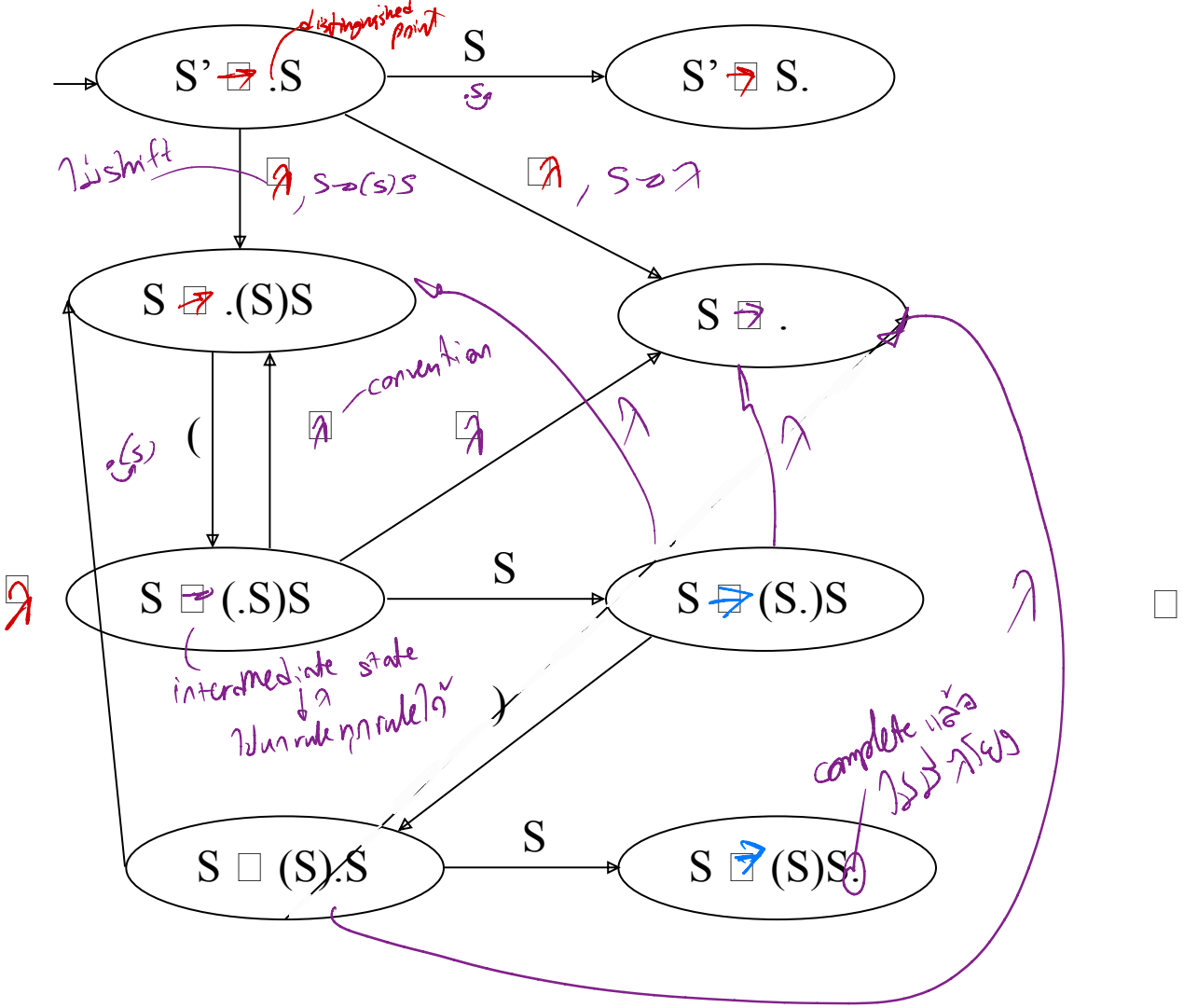
Finite automata of items

Grammar:

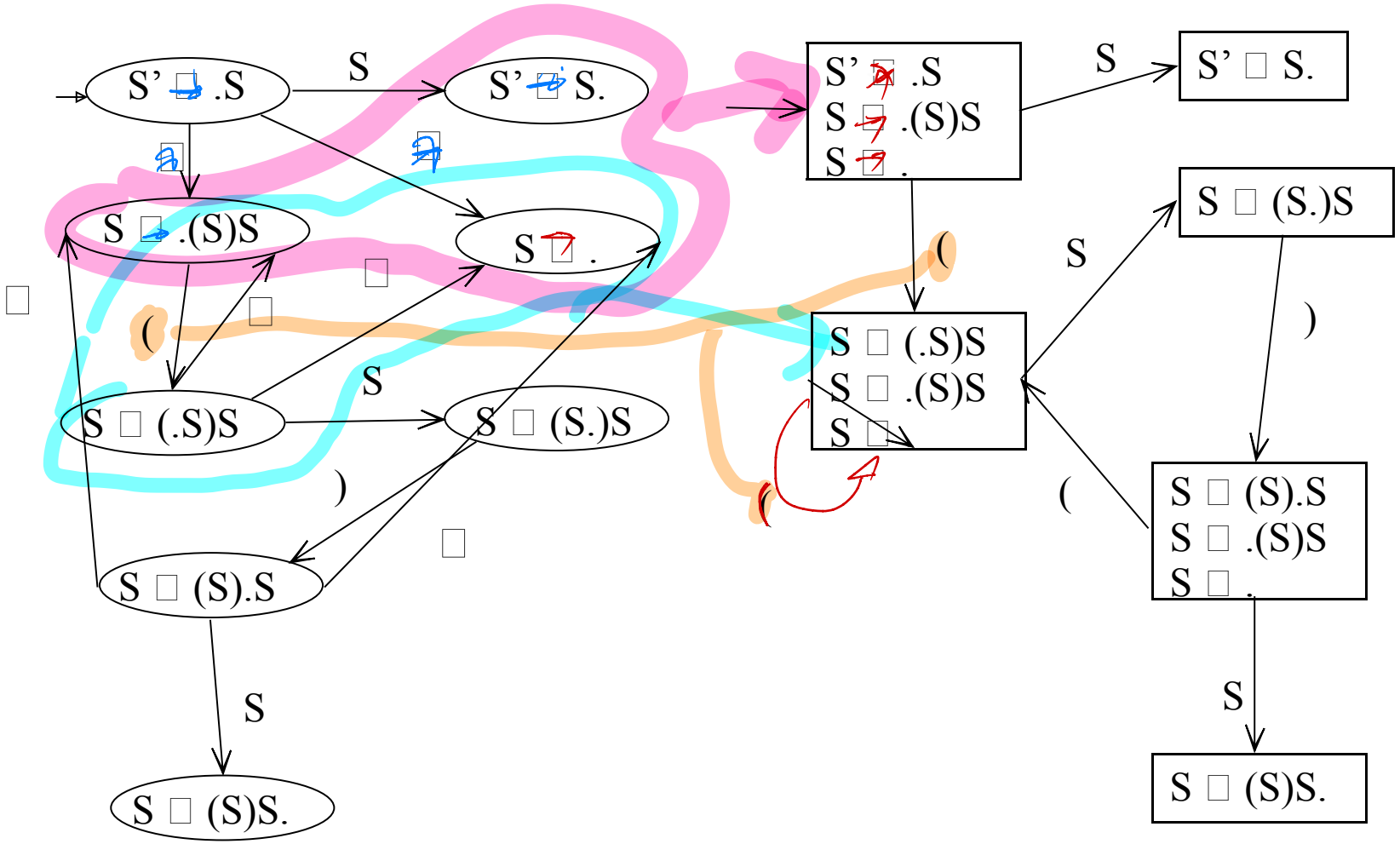
1. S' → S
2. S → (S)S
3. S → λ

Items:

1. S' → .S
2. S' → S.
3. S → .(S)S
4. S → (.S)S
5. S → (S.)S
6. S → (S).S
7. S → (S)S.
8. S → .



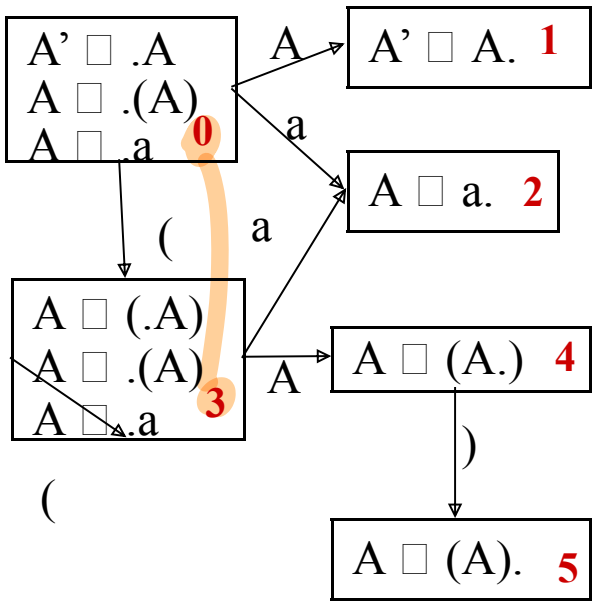
DFA of LR(0) Items



LR(0) parsing algorithm

Item in state	next token	Action
A-> x.By where B is terminal	B	shift B and push state s containing A -> xB.y
A-> x.By where B is terminal	not B	error
A -> x.	-	reduce with A -> x (i.e. pop x, backup to the state s on top of stack) and push A with new state d(s,A)
S' -> S.	none	accept
S' -> S.	any	error

LR(0) Parsing Table



State	Action	Rule	(	a	)	A
0	shift		3	2		1
1	reduce	A' -> A				
2	reduce	A -> a				
3	shift		3	2		4
4	shift				5	
5	reduce	A -> (A)				

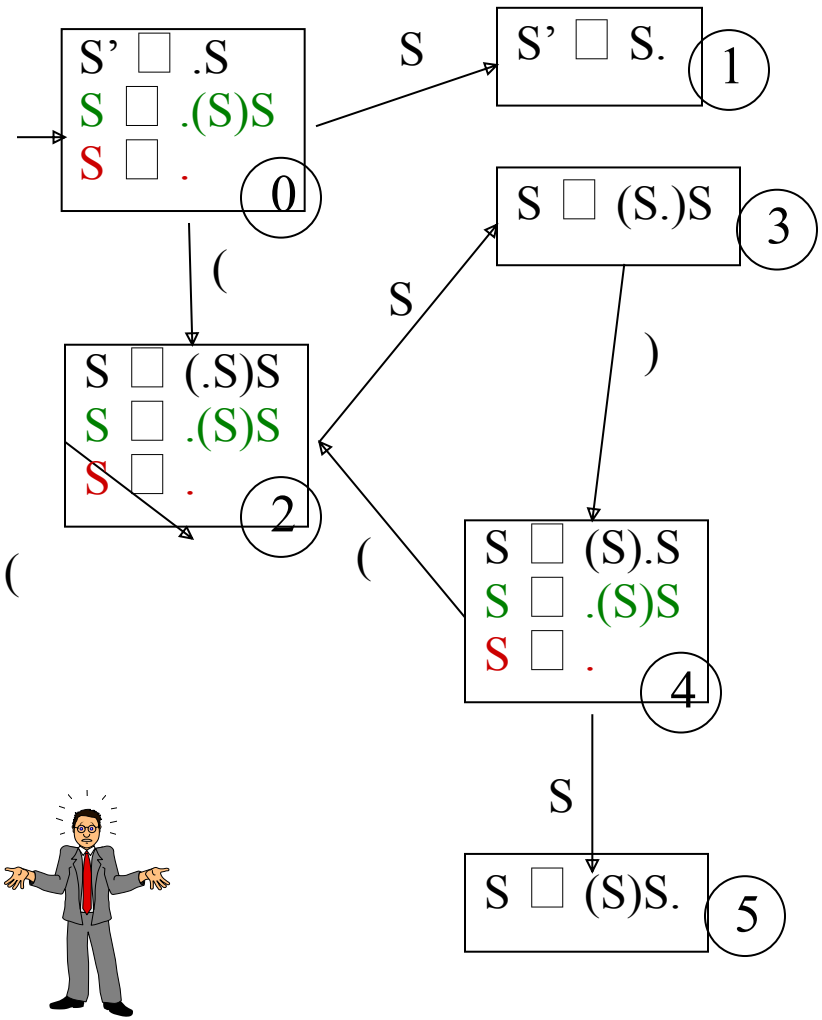
Example of LR(0) Parsing

State	Action	Rule	(	a	)	A
0	shift		3	2		1
1	reduce	A' -> A				
2	reduce	A -> a				
3	shift		3	2		4
4	shift				5	
5	reduce	A -> (A)				

Stack	Input	Action
$0	((a)) $	shift
$0(3	(a)) $	shift
$0(3(3	a)) $	shift
$0(3(3a2	)) $	reduce
$0(3(3A4	)) $	shift
$0(3(3A4)5	) $	reduce
$0(3A4	) $	shift
$0(3A4)5	$	reduce
$0A1	$	accept

Non-LR(0)Grammar

- Conflict
  - Shift-reduce conflict
    - A state contains a complete item A → x. and a shift item A → x.By
  - Reduce-reduce conflict
    - A state contains more than one complete items.
- A grammar is a LR(0) grammar if there is no conflict in the grammar.



SLR(1) parsing

- Simple LR with 1 lookahead symbol
- Examine the next token before deciding to shift or reduce
  - If the next token is the token expected in an item, then it can be shifted into the stack.
  - If a complete item A □ x. is constructed and the next token is in Follow(A), then reduction can be done using A □ x.
  - Otherwise, error occurs.
- Can avoid conflict

SLR(1) parsing algorithm

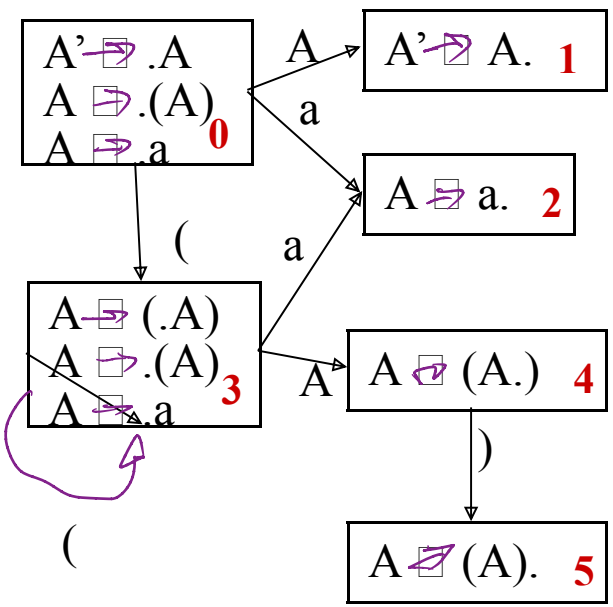
Item in state	token	Action
A-> x.By (B is terminal)	B	shift B and push state s containing A -> xB.y
A-> x.By (B is terminal)	not B	error
A -> x.	in Follow(A)	reduce with A -> x (i.e. pop x, backup to the state s on top of stack) and push A with new state d(s,A)
A -> x.	not in Follow(A)	error
S' -> S.	none	accept
S' -> S.	any	error

SLR(1) grammar

- Conflict
  - Shift-reduce conflict
    - A state contains a shift item A □ x.Wy such that W is a terminal and a complete item B □ z. such that W is in Follow(B).
  - Reduce-reduce conflict
    - A state contains more than one complete item with some common Follow set.
- A grammar is an SLR(1) grammar if there is no conflict in the grammar.

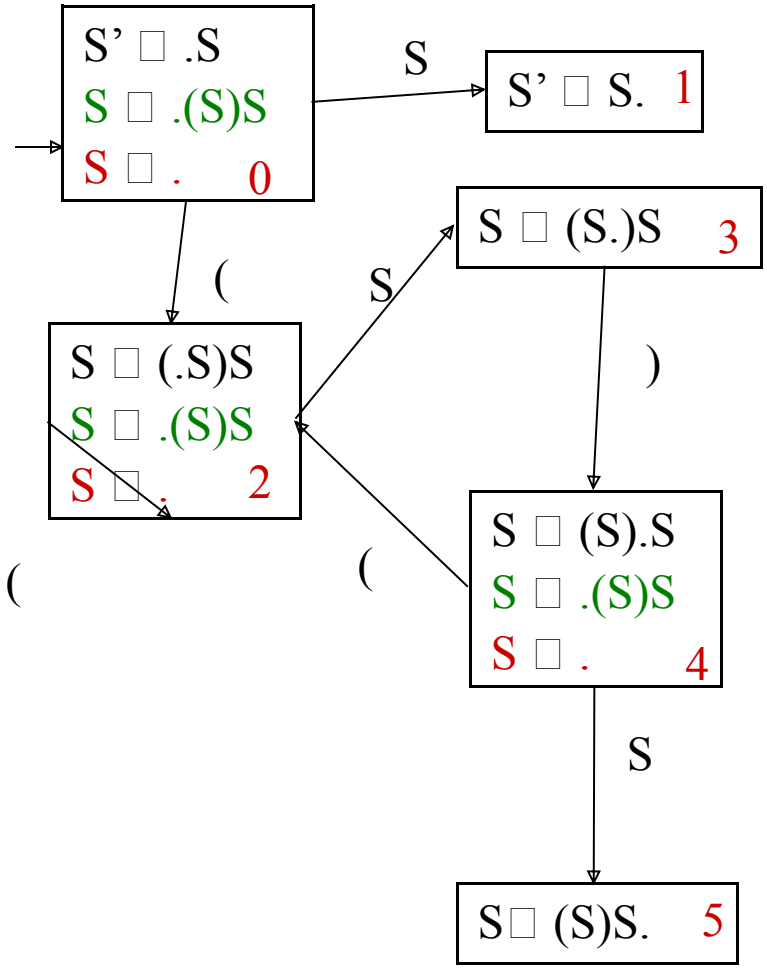
SLR(1) Parsing Table

A → (A) | a



State	(	a	)	$	A
0	S3	S2			1
1				AC	
2			R2		
3	S3	S2			4
4			S5		
5			R1		

SLR(1) Grammar not LR(0)



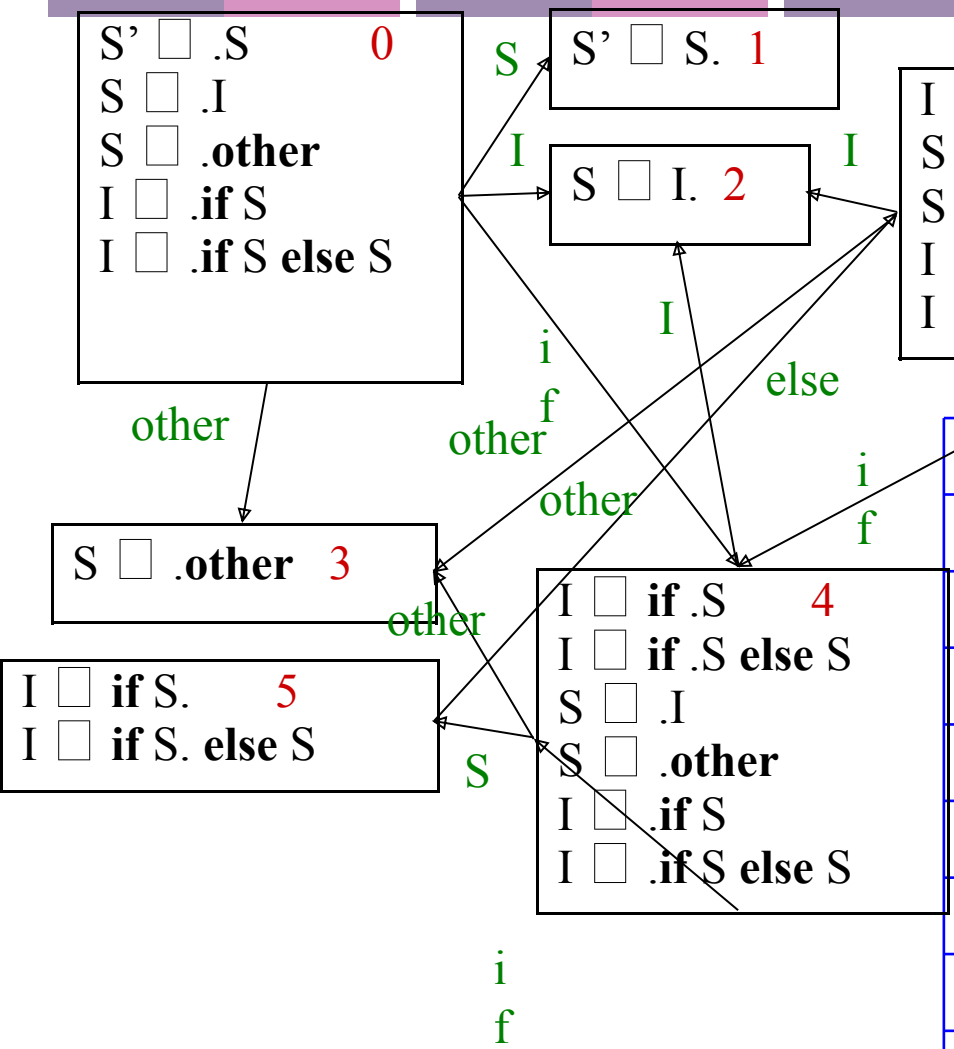
S □ (S)S | □

State	(	)	$	S
0	S2	R2	R2	1
1			AC	
2	S2	R2	R2	3
3		S4		
4	S2	R2	R2	5
5		R1	R1	

Disambiguating Rules for Parsing Conflict

- Shift-reduce conflict
  - Prefer shift over reduce
    - In case of nested if statements, preferring shift over reduce implies most closely nested rule for dangling else
- Reduce-reduce conflict
  - Error in design

Dangling Else



state	if	else	other	$	S	I
0	S4		S3		1	2
1				AC		
2		R1		R1		
3		R2		R2		
4	S4		S3		5	2
5		S6		R3		
6	S4		S3		7	2
7		R4		R4		

End