Parsing

Jaruloj Chongstitvatana
Department of Mathematics and Computer Science
Chulalongkorn University

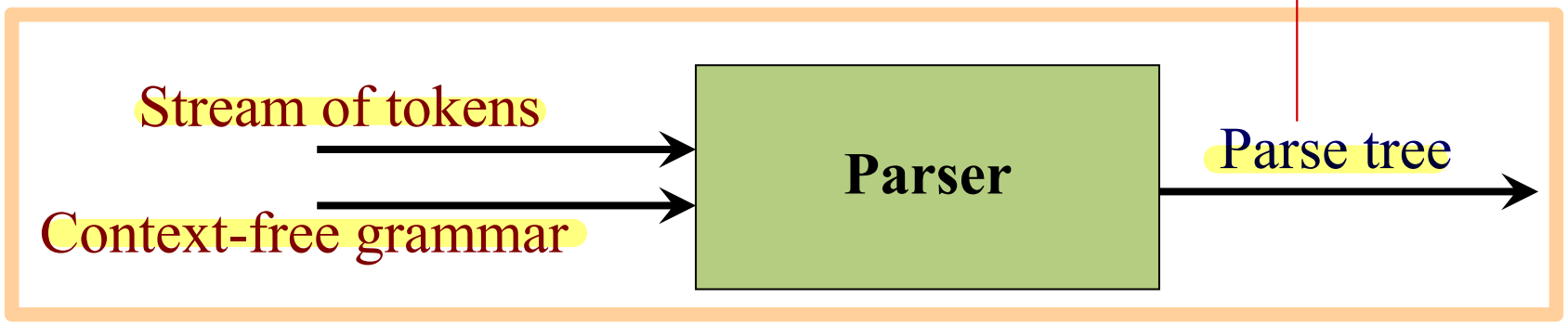
Outline

* ส่วนใหญ่ใช้อันนี้

- 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

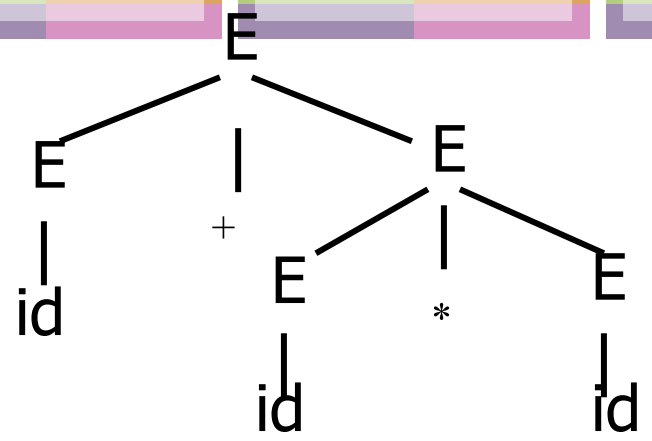
Try different structures and backtrack if it does not matched the input

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

Parse Trees and Derivations



Top-down parsing

ราก

ใบ

$E \Rightarrow E + E$

$\Rightarrow id + E$

$\Rightarrow id + E * E$

$\Rightarrow id + id * E$

$\Rightarrow id + id * id$

leftmost derivation

=

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.

เลือกตัวไหน!
เลือกกฎไหน!

(เกาล่วงหน้า)

Top-down Parsing

- Why is it difficult?
  - Cannot decide until later (ตัดสินใจเกี่ยวนี้ไม่ได้)
    - 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 |...

    เช่น if บางที่ตามมีหลาย step หลายขั้นก่อนถึง else
  - Production with empty string
    - Next token: **id** Structure to be built: par
    - par → parList | λ
    - parList → exp , parList | exp

Recursive-Descent

(กฎเกณฑ์การเขียน)

- 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.

มี grammar เราต้องเขียน parser เพื่อดูว่าสมกับ grammar ใหม่

Recursive-Descent: Example

Ex $E \rightarrow E\ O\ F \mid F$

$O \rightarrow + \mid -$

$F \rightarrow (\ E\ ) \mid id$

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; }
```

non-terminal

ปรับไม่ขึ้นเป็น infinite loop)

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

ต้องพยายามไม่ให้มี left recursion

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

token เป็น +

Extended Backus-Naur Form (EBNF)

- Kleene's Star/ Kleene's Closure
  - Seq ::= St {; St}
  - Seq ::= {St ;} St
- Optional Part
  - IfSt ::= if (E) St [else St]
  - E ::= F [+ E] | F [- E]

มี/ไม่มีก็ได้

Match procedure

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

check ว่า token ที่เข้ามา ตรงกับ token ที่เราต้องการไหม

- 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 λ-production
  $A \rightarrow \lambda$

ตัดสินใจไม่ได้ จับไหม? ไปต่อ?

A recursive descent parser for ASM language

Let us begin with defining a language, a simple assembly language (invent by Tawan). We show a sample of the code.

```
CLRA
MOV R1,100
MOV R2,200
ADD R1
ADD R2
MOVA R1
```

เขียนเป็น grammar

Grammar of ASM

```
asm = op oprnd asm | EOF
op = CLRA | MOV | ADD | MOVA
oprnd = reg , num | reg | empty
reg = R[0..31]
```

เขียน Parser

Parser

return 0 - error, 1 - OK

```
reg()
  match('R')
  ret num()

op()
  switch tokentype()
    mov: match('MOV')
    ...
    default: ret 0

oprnd()
  if reg() != 0
     if token == ','   // lookahead
        match(',')
        ret num()
     else
        ret 1
   ret 1

asm()
   if token == EOF  ret 1
   if op() != 0
      oprnd()
      asm()
   ret 0
```

LL(1) Parsing

เป็น Automation ที่ทำงาน parse อย่าง

- 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 $\rightarrow$ Y.
  - Which production will be chosen, if there are both X $\rightarrow$ Y and X $\rightarrow$ Z ?

คลิปที่ 11 20:00 (animation)

Example of LL(1) Parsing

เป็น starting symbol ∴ใส่ลงไปก่อนเลย

$E \Rightarrow TX$ ← pop E ออก เสียใส่ TX เข้าไป
$\Rightarrow FNX$ เลือกเป็น (E) เพราะใน input เรามีแบบนี้เลย
$\Rightarrow (E)NX$ เราจะ pop F ออก เสียใส่
$\Rightarrow (TX)NX$
$\Rightarrow (FNX)NX$
$\Rightarrow (nNX)NX$ เพราะ input เป็น n
$\Rightarrow (nX)NX$
$\Rightarrow (nATX)NX$
$\Rightarrow (n+TX)NX$
$\Rightarrow (n+FNX)NX$
$\Rightarrow (n+(E)NX)NX$
$\Rightarrow (n+(TX)NX)NX$
$\Rightarrow (n+(FNX)NX)NX$
$\Rightarrow (n+(nNX)NX)NX$
$\Rightarrow (n+(nX)NX)NX$
$\Rightarrow (n+(n)NX)NX$
$\Rightarrow (n+(n)X)NX$
$\Rightarrow (n+(n))NX$
$\Rightarrow (n+(n))MFNX$
$\Rightarrow (n+(n))*FNX$
$\Rightarrow (n+(n))*nNX$
$\Rightarrow (n+(n))*nX$
$\Rightarrow (n+(n))*n$

top of stack เป็น terminal และตรงกับ input ตัวที่เพดี ∴ pop ออก และขยับ pr

ใช้กฎไหนดี!!? (เช่น MFN หรือ λ) ดีเทคต่อจะเจอ * ซึ่งไม่ตรงกับ input ∴ จึงตัดไป λ

n
N
X
)
N
*
n
N
X
$

Finished

ตำแหน่งสุดของ stack

1 token

dollar sign = end of input

(	n	+	(	n	)	)	*	n	$
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
pr									

ขยับไปเรื่อยๆจนหมด

start symbol

$E \rightarrow TX$
$X \rightarrow ATX \mid \lambda$
$A \rightarrow + \mid -$
$T \rightarrow FN$
$N \rightarrow MFN \mid \lambda$
$M \rightarrow *$
$F \rightarrow (E) \mid n$

grammar

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 $\rightarrow X_1\ X_2\ \ldots\ X_n$ from the parsing table entry M[A, a] Pop stack;

Push $X_n\ \ldots\ X_2\ X_1$ into stack in that order

ELSE Error

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?

เลือกไงดี!

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

t
Y
Q

X
t
Y

first set = set ของ terminal symbol ที่ได้จากกฎข้อนั้น

left deviation ที่เรียน

t	...	...	...

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 $X \rightarrow X_1 X_2 \ldots X_n$ is a rule, then
    - First(X_1) -{λ} is a subset of First(X)
    - First(X_i)-{λ} is a subset of First(X) if for all $j<i$ First(X_j) contains {λ}
    - λ is in First(X) if for all $j \leq n$ First(X_j)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 | λ

exp → 0 | 1

* ตามปกติ ตัวชี้ขาดสุดเป็น terminal (ถ้าหาในบรรทัดอื่นๆ ก็ด้วย)

First(exp) = {0,1}

First(elsepart) = {else, λ}

First(ifst) = {if}

First(st) = {if, other}

if จาก First(ifst)

first(factor) ก็จัดว่าเหมือน first(term)

ทำนองเดียวกัน

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 $\rightarrow X_1 X_2 \ldots X_n$

    For each X_i in $\{X_1, X_2, \ldots, X_n\}$

      If for all $j<i$ First(X_j) contains λ,

      Then

        add First(X_i)-{λ} to First(A)

  If λ is in First(X_1), First(X_2), ..., and First(X_n)

  Then add λ to First(A)

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

If A is a nonterminal, then for each rule A $\rightarrow X_1 X_2 \ldots X_n$, First(A) contains First(X_1) - {λ}.

If also for some $i<n$, First(X_1), First(X_2), ..., and First(X_i) contain λ, then First(A) contains First(X_{i+1})-{λ}.

If First(X_1), First(X_2), ..., and First(X_n) contain λ, then First(A) also contains λ.

Finding First Set: An Example

exp → term exp′

exp′ → addop term exp′ | λ

addop → + | -

term → factor term′

term′ → mulop factor term′ | λ

mulop → *

factor → (exp) | num

	First
exp	
exp′	
addop	
term	
term′	
mulop	
factor	

Finding First Set: An Example

exp → term exp′

exp′ → addop term exp′ | λ

addop → + | -

term → factor term′

term′ → mulop factor term′ | λ

mulop → *

factor → (exp) | num

	First
exp	(num
exp′	+ - λ
addop	+ -
term	(num
term′	λ *
mulop	*
factor	(num

* เอาเครื่องหมาย

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 $\rightarrow$ X A Y, then First(Y) - {λ} is in Follow(A).
- If there is production B $\rightarrow$ 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)={}

Initialize

WHILE change is made to some Follow sets

FOR each production $A \rightarrow X_1 X_2 \ldots X_n$,

FOR each nonterminal X_i

Add First($X_{i+1} X_{i+2} \ldots X_n$)-{λ} into Follow($X_i$).

(NOTE: If i=n, $X_{i+1} X_{i+2} \ldots X_n = \lambda$)

IF λ is in First($X_{i+1} X_{i+2} \ldots X_n$) THEN

Add Follow(A) to Follow(X_i)

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

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

If there is production $B \rightarrow X A Y$ and λ is in First(Y), then Follow(A) contains Follow(B).

Finding Follow Set: An Example

exp → term exp′

exp′ → addop term exp′ | λ

addop → + | -

term → factor term′

term′ → mulop factor term′ |λ

mulop → *

factor → (exp) | num

	First	Follow
exp	(num	$)
exp′	λ + -	$)
addop	+ -	(num
term	(num	+ - $)
term′	λ *	+ - $)
mulop	*	(num
factor	(num	* + - $)

Computing Follow

1. If A is start symbol, put $ in FOLLOW(A)
2. Productions of the form B → α A β,
   FOLLOW(A)= FIRST(β)
3. Productions of the form B → α A or
   B → α A β where β ⇒* ε
   Add FOLLOW(A)= FOLLOW(B)

สไลด์ที่แล้ว

1. Place $ in FOLLOW(S), where S is the start symbol and $ is the input right endmarker.
2. If there is a production A ⇒ αBβ, then everything in FIRST(β), except for ε, is placed in FOLLOW(B).
3. If there is a production A ⇒ αB, or a production A ⇒ αBβ where FIRST(β) contains ε (i.e., β ⇒ε), then everything in FOLLOW(A) is in FOLLOW(B).

Rules to compute FOLLOW set:

1) FOLLOW(S) = { $ } // where S is the starting Non-Terminal
2) If A -> pBq is a production, where p, B and q are any grammar symbols, then everything in FIRST(q) except ε is in FOLLOW(B).
3) If A->pB is a production, then everything in FOLLOW(A) is in FOLLOW(B).
4) If A->pBq is a production and FIRST(q) contains ε, then FOLLOW(B) contains { FIRST(q) – ε } U FOLLOW(A)

exp → term exp′

exp′ → addop term exp′ | λ

addop → + | -

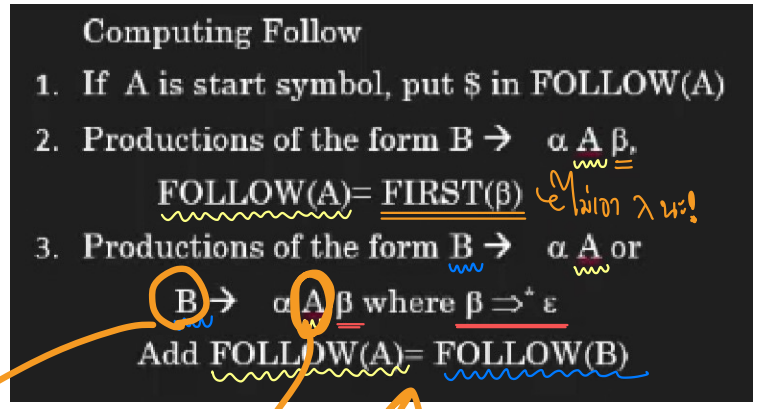
term → factor term′

term′ → mulop factor term′ |λ

mulop → *

factor → (exp) | num

พิจารณา



α เป็นอะไรก็ได้

exp′ → α A addop **B** term exp′ | λ

Follow(addop) = First(term) ตามกฎข้อ 2

exp′ → α **A** term **B′** exp′ | λ

B′ → λ

Follow(term) = First(exp′) ∪ Follow(exp′)

ตามกฎข้อ 2 ตามกฎข้อ 3

เพราะ exp′ เป็น ε ได้

A → B | λ

exp → term exp′

exp′ → addop term exp′ | λ

addop → + | -

term → factor term′

term′ → mulop factor term′ |λ

mulop → *

factor → (exp) | num

Computing Follow

1. If A is start symbol, put $ in FOLLOW(A)
2. Productions of the form B → α A β,
   FOLLOW(A)= FIRST(β) (ยกเว้น λ นะ!)
3. Productions of the form B → α A or
   B → α A β where β ⇒* ε
   Add FOLLOW(A)= FOLLOW(B)

Follow(exp) = {), $ }

Follow(exp′) = Follow(exp) = {), $ }

** Follow(addop) = First(term) = { (, num }

Follow(term) = First(exp′) ∪ Follow(exp′) = { +, -,), $ } ({+, -})

Follow(term′) = Follow(term) = { +, -,), $ }

** Follow(mulop) = First(factor) = { (, num }

Follow(factor) = First(term′) ∪ Follow(term′) ({*})

= { *, +, -,), $ }

Constructing LL(1) Parsing Tables

FOR each nonterminal A and a production A $\rightarrow$ X

FOR each token a in First(X)

A $\rightarrow$ X is in M(A, a)

IF λ is in First(X) THEN

FOR each element a in Follow(A)

Add A $\rightarrow$ X to M(A, a)

มีทางไปเจอ First set ถ้าจุดที่ต้องตัดสินใจ → ใช้ following set

Example: Constructing LL(1) Parsing Table

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

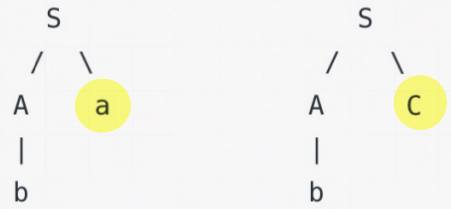
1 exp → term exp'
2 exp' → addop term exp'
3 exp' → λ
4 addop → +
5 addop → -
6 term → factor term'
7 term' → mulop factor term'
8 term' → λ
9 mulop → *
10 factor → (exp)
11 factor → num

	(	)	+	-	*	n	$
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	

Follow(X) to be the set of terminals that can appear immediately to the right of Non-Terminal X in some sentential form.

Example:

```
S ->Aa | Ac
A ->b
```



Here, FOLLOW (A) = {a, c}

Rules to compute FOLLOW set:

1) FOLLOW(S) = { $ } // where S is the starting Non-Terminal

2) If A -> pBq is a production, where p, B and q are any grammar symbols, then everything in FIRST(q) except Є is in FOLLOW(B).

3) If A->pB is a production, then everything in FOLLOW(A) is in FOLLOW(B).

4) If A->pBq is a production and FIRST(q) contains Є, then FOLLOW(B) contains { FIRST(q) - Є } U FOLLOW(A)

Example 1:

Production Rules:

① E -> TE'
② E' -> +T E'|Є
③ T -> F T'
④ T' -> *F T' | Є
⑤ F -> (E) | id

FIRST set

FIRST(E) = FIRST(T) = { (, id }
FIRST(E') = { +, Є }
FIRST(T) = FIRST(F) = { (, id }
FIRST(T') = { *, Є }
FIRST(F) = { (, id }

FOLLOW Set

meaning follow ')'

FOLLOW(E) = { $,) } // Note ')' is there because of 5th rule
FOLLOW(E') = FOLLOW(E) = { $,) } // See 1st production rule
FOLLOW(T) = { FIRST(E') - Є } U FOLLOW(E') U FOLLOW(E) = { + , $,) }
FOLLOW(T') = FOLLOW(T) = { + , $,) }
FOLLOW(F) = { FIRST(T') - Є } U FOLLOW(T') U FOLLOW(T) = { *, +, $,) }

Example 2:

```
Production Rules:
S -> aBDh
B -> cC
C -> bC | Є
D -> EF
E -> g | Є
F -> f | Є

FIRST set
FIRST(S) = { a }
FIRST(B) = { c }
FIRST(C) = { b , Є }
FIRST(D) = FIRST(E) U FIRST(F) = { g, f, Є }
FIRST(E) = { g , Є }
FIRST(F) = { f , Є }

FOLLOW Set
FOLLOW(S) = { $ }
FOLLOW(B) = { FIRST(D) – Є } U FIRST(h) = { g , f , h }
FOLLOW(C) = FOLLOW(B) = { g , f , h }
FOLLOW(D) = FIRST(h) = { h }
FOLLOW(E) = { FIRST(F) – Є } U FOLLOW(D) = { f , h }
FOLLOW(F) = FOLLOW(D) = { h }
```

Example 3:

```
Production Rules:
S -> ACB|Cbb|Ba
A -> da|BC
B-> g|Є
C-> h| Є

FIRST set
FIRST(S) = FIRST(A) U FIRST(B) U FIRST(C) = { d, g, h, Є, b, a}
FIRST(A) = { d } U {FIRST(B)–Є} U FIRST(C) = { d, g, h, Є }
FIRST(B) = { g, Є }
FIRST(C) = { h, Є }

FOLLOW Set
FOLLOW(S) = { $ }
FOLLOW(A)  = { h, g, $ }
FOLLOW(B) = { a, $, h, g }
FOLLOW(C) = { b, g, $, h }
```

Note :

1. Є as a FOLLOW doesn't mean anything (Є is an empty string).
2. $ is called end-marker, which represents the end of the input string, hence used while parsing to indicate that the input string has been completely processed.
3. The grammar used above is Context-Free Grammar (CFG). The syntax of a programming language can be specified using CFG.
4. CFG is of the form A -> B, where A is a single Non-Terminal, and B can be a set of grammar symbols (i.e. Terminals as well as Non-Terminals)

https://www.cp.eng.chula.ac.th/~prabhas//teaching/prola...

Example: Constructing LL(1) Parsing Table

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

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

	(	)	+	-	*	n	$
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	

25

LL1-parser-explain - Notepad

File Edit Format View Help

Pseudo-Program

```
step

push starting symbol #exp

rule 1   exp
  pop
  push #exp2
  push #term

rule 2   exp2-2
  pop
  push #exp2
  push #term
  push #addop

rule 3  exp2-3
  pop

rule 4  addop-4
  pop
  gettoken
```

push ตัวที่อยู่ก่อน (จะได้อยู่ล่างสุด)

(ตัวอื่นๆ ดูใน คลิปที่ 11)

Ln 1, Col 1 120% Windows (CRLF) UTF-8

ENG 10:33 AM 4/14/2020

```
push #exp

loop until tos = $
{
x = look up next rule (tos,current-token)

switch x
  0 : error()
  1 : exp()
  2 : exp2-2()
  3 : exp2-3()
  4 : addop-4()
  5 : addop-5()
  6 : term()
  7 : term2-7()
  8 : term2-8()
  9 : mulop()
  10 : factor-10()
  11 : factor-11()
  12 : rparen()
}
```

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 → *

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

ไปได้หลายทาง ถ้าวน! (ต้องแก้ที่ grammar)

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

Causes of Non-LL(1) Grammar

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

Left Recursion

- Immediate left recursion (ทำได้เฉพาะไม่มี λ)
  - $A \rightarrow A\ X \mid Y$
  - $A \rightarrow A\ X_1 \mid A\ X_2 \mid \ldots \mid A\ X_n \mid Y_1 \mid Y_2 \mid \ldots \mid Y_m$

เห็นให้เป็น right recursion

$A = Y\ X^*$

$A = \{Y_1, Y_2, \ldots, Y_m\}\ \{X_1, X_2, \ldots, X_n\}^*$

- Can be removed very easily
  - $A \rightarrow Y\ A',\ A' \rightarrow X\ A' \mid \lambda$
  - $A \rightarrow Y_1\ A' \mid Y_2\ A' \mid \ldots \mid Y_m\ A',\ A' \rightarrow X_1\ A' \mid X_2\ A' \mid \ldots \mid X_n\ A' \mid \lambda$

- General left recursion
  - $A \Rightarrow X \Rightarrow^* A\ Y$

- Can be removed when there is no empty-string production and no cycle in the grammar

ไม่เห็นมาก)

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 exp′ | - term exp′ | λ

term → factor term′

term′ → * factor term′ | λ

factor → (exp) | num

exp = term (± term)*

term = factor (* factor)*

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

Left Factoring

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

$A \rightarrow X\ Y \mid X\ Z$

can be left-factored as

$A \rightarrow X\ A'$ and $A' \rightarrow Y \mid Z$

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 | λ

seq → st ; seq | st

can be left-factored as

seq → st seq′

seq′ → ; seq | λ

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 $A \rightarrow B$

Example of Shift-reduce Parsing

- Grammar

  $S' \rightarrow S$

  $S \rightarrow (S)S \mid \lambda$

- Parsing actions

Stack	Input	Action
$	(( )) $	
$ (	( )) $	
$ ((	)) $	
$ ((S	)) $	
$ ((S)	) $	
$ ((S) S	) $	
$ (S	) $	
$ (S)	$	
$ (S) S	$	
$ S	$	

- Reverse of
- rightmost derivation
- from left to right

1 $\Rightarrow$ **(( ))**
2 $\Rightarrow$ **(( ))**
3 $\Rightarrow$ **(( ))**
4 $\Rightarrow$ **((S))**
5 $\Rightarrow$ **((S))**
6 $\Rightarrow$ **((S) S)**
7 $\Rightarrow$ **(S)**
8 $\Rightarrow$ **(S)**
9 $\Rightarrow$ **(S) S**
10 **S'** $\Rightarrow$ **S**

Example of Shift-reduce Parsing

- Grammar
  - $S' \rightarrow S$
  - $S \rightarrow (S)S \mid \lambda$
- Parsing actions

Stack	Input	Action
$	(( )) $	shift
$ (	( )) $	shift
$ ((	)) $	reduce $S \rightarrow \lambda$
$ ((S	)) $	shift
$ ((S)	) $	reduce $S \rightarrow \lambda$
$ ((S) S	) $	reduce $S \rightarrow (S) S$
$ (S	) $	shift
$ (S)	$	reduce $S \rightarrow \lambda$
$ (S) S	$	reduce $S \rightarrow (S) S$
$ S	$	accept

Viable prefix

1	$\Rightarrow$ (( ))	handle
2	$\Rightarrow$ (( ))	
3	$\Rightarrow$ (( ))	
4	$\Rightarrow$ ((S))	
5	$\Rightarrow$ ((S))	
6	$\Rightarrow$ ((S) S)	
7	$\Rightarrow$ (S)	
8	$\Rightarrow$ (S)	
9	$\Rightarrow$ (S) S	
10 S′	$\Rightarrow$ S	

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 $S \rightarrow \lambda$
  - (S) S . with $S \rightarrow \lambda$
  - ((S) S .) with $S \rightarrow$ (S) S
- LR(0) item
  - $S \rightarrow$ (S) S.
  - $S \rightarrow$ (S) . S
  - $S \rightarrow$ (S .) S
  - $S \rightarrow$ (. S) S
  - $S \rightarrow$. (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′ $\rightarrow$ 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 $A \rightarrow w \,.\, B\ y$ means:
    - $A \rightarrow w\ B\ y$ 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 $A \rightarrow w\ B \,.\, Y$

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 λ-transition
- Kernel Item
  - Original item, not including closure items

Finite automata of items

Grammar:

$S' \rightarrow S$

$S \rightarrow (S)S$

$S \rightarrow \lambda$

Items:

$S' \rightarrow .S$

$S' \rightarrow S.$

$S \rightarrow .(S)S$

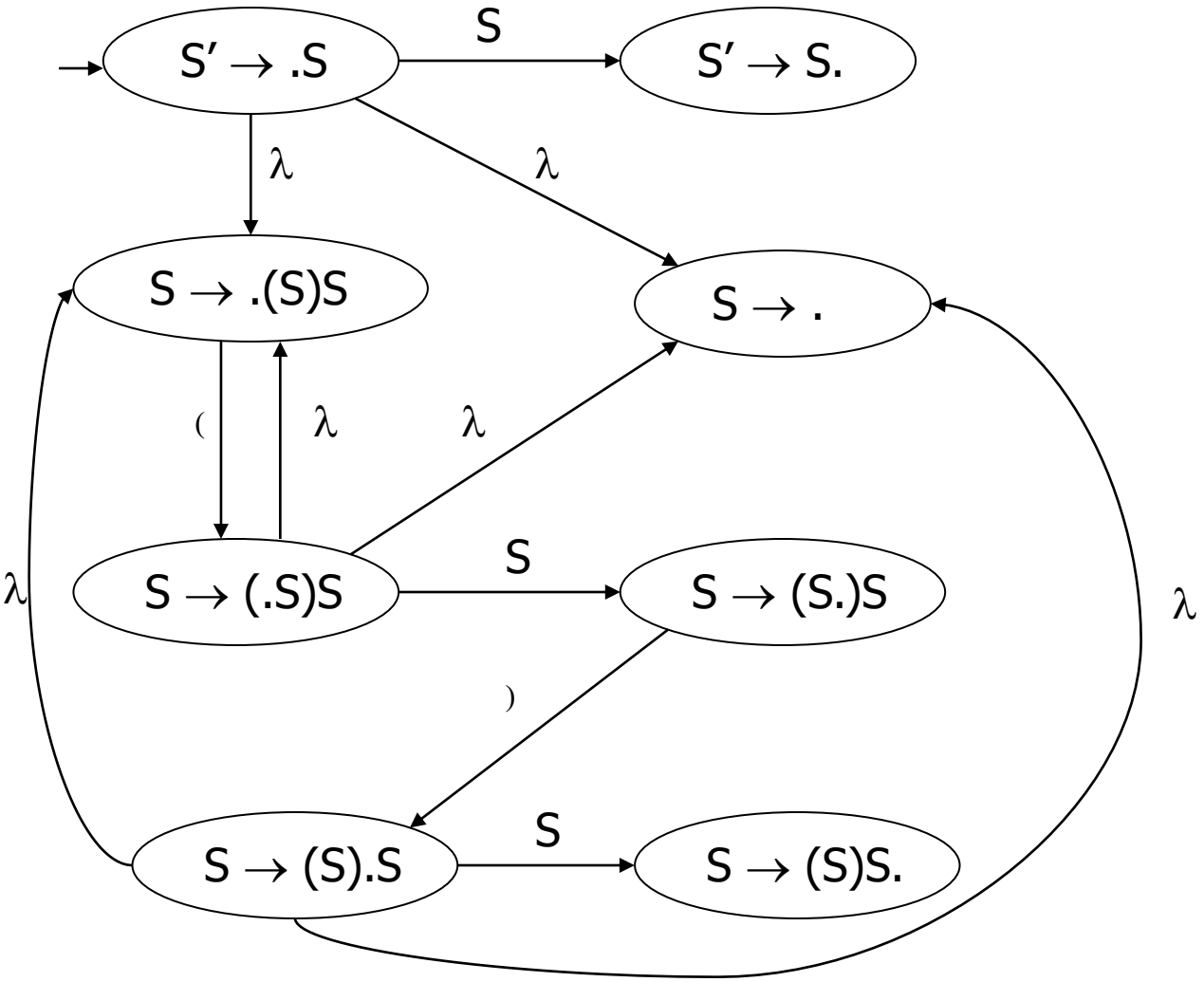
$S \rightarrow (.S)S$

$S \rightarrow (S.)S$

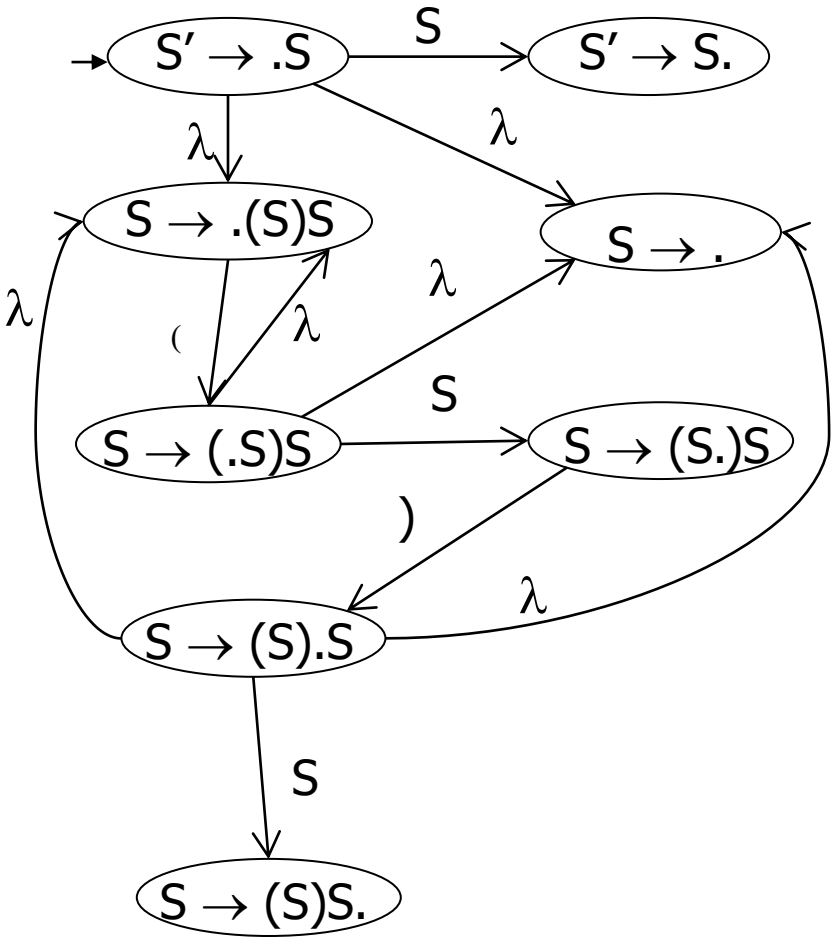
$S \rightarrow (S).S$

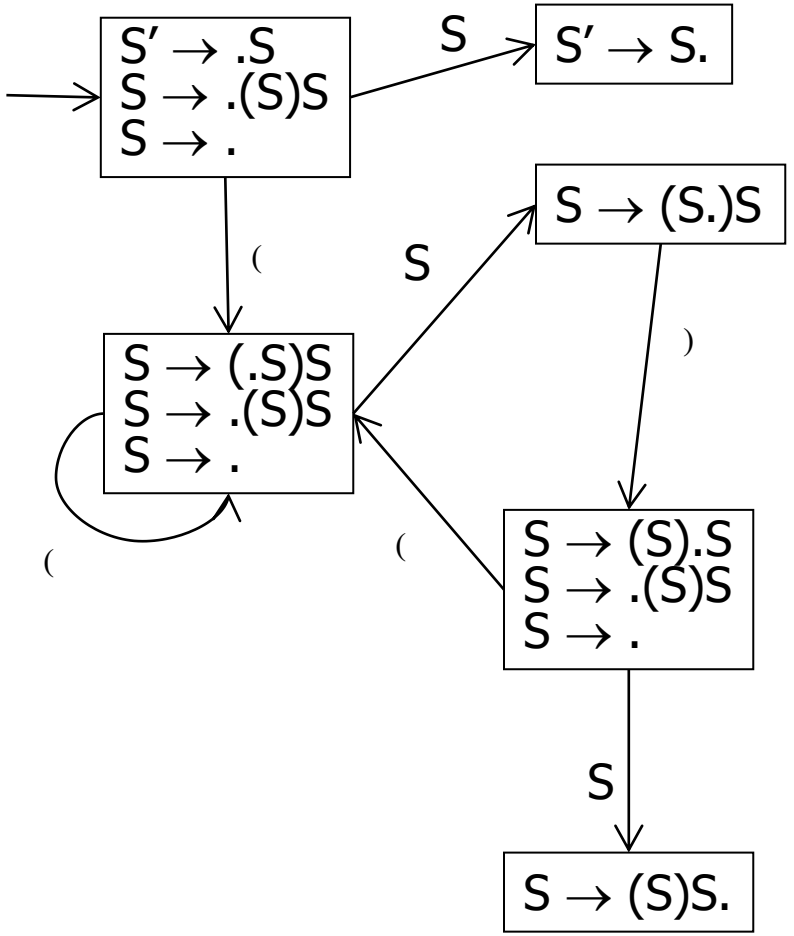
$S \rightarrow (S)S.$

$S \rightarrow .$



DFA of LR(0) Items

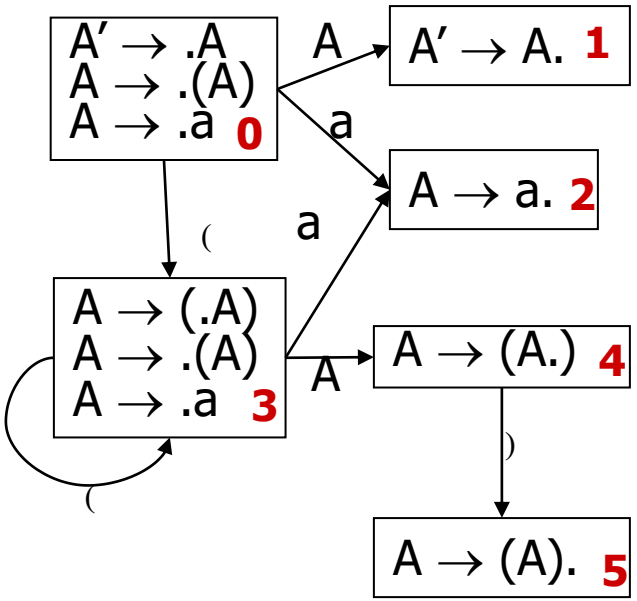




LR(0) parsing algorithm

Item in state	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)				

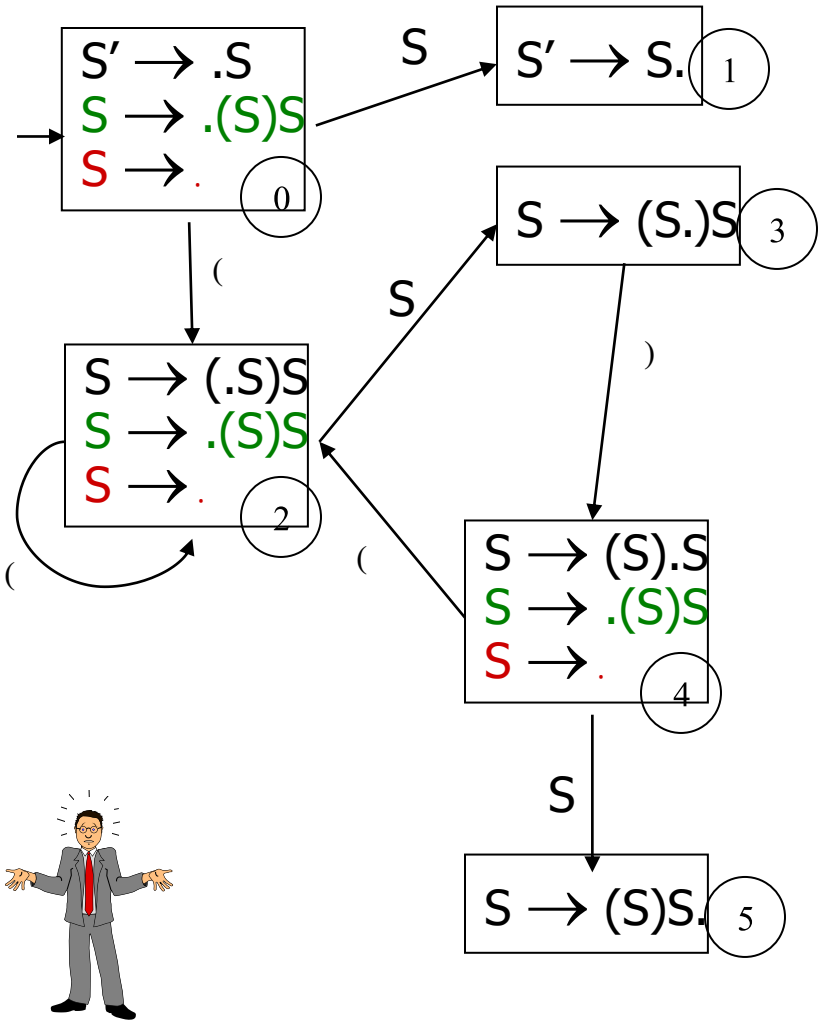
Example of LR(0) Parsing

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 $A \rightarrow x.$ is constructed and the next token is in Follow(A), then reduction can be done using $A \rightarrow x$.
  - Otherwise, error occurs.
- Can avoid conflict

SLR(1) parsing algorithm

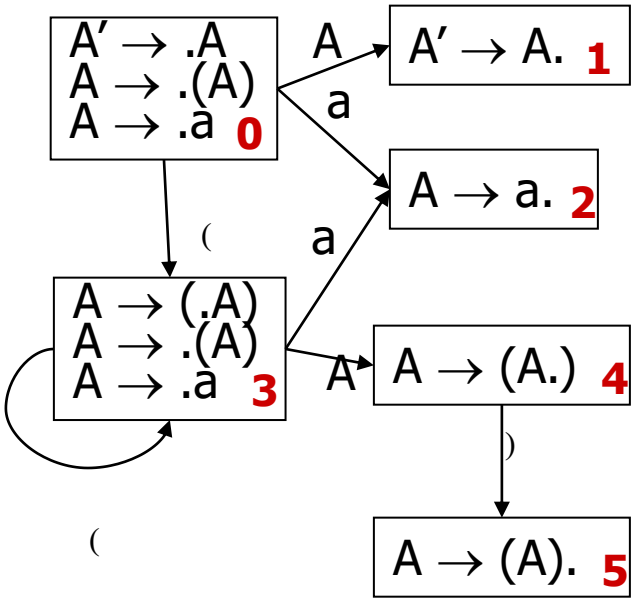
Item in state	token	Action
A-> x.By (B is terminal)	B	shift B and push state s containing A -> xB.y
A-> x.By (B is terminal)	not B	error
A -> x.	in Follow(A)	reduce with A -> x (i.e. pop x, backup to the state s on top of stack) and push A with new state d(s,A)
A -> x.	not in Follow(A)	error
S’ -> S.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

SLR(1) grammar

- Conflict
  - Shift-reduce conflict
    - A state contains a shift item $A \rightarrow x.Wy$ such that W is a terminal and a complete item $B \rightarrow z.$ 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.

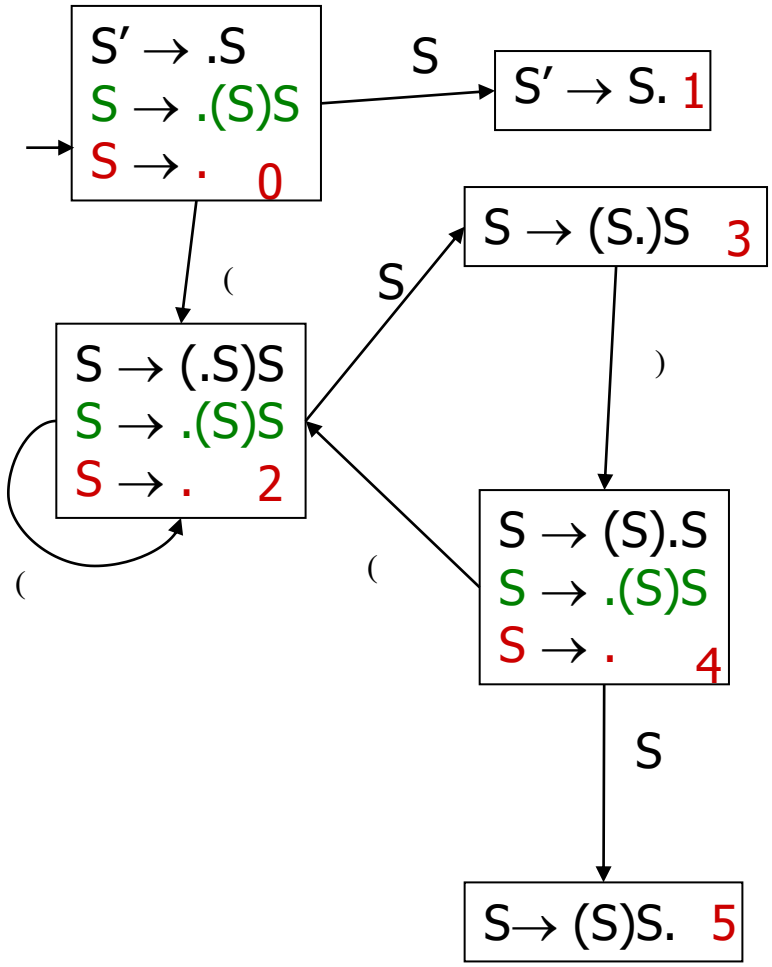
SLR(1) Parsing Table

$A \rightarrow (A) \mid a$



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)



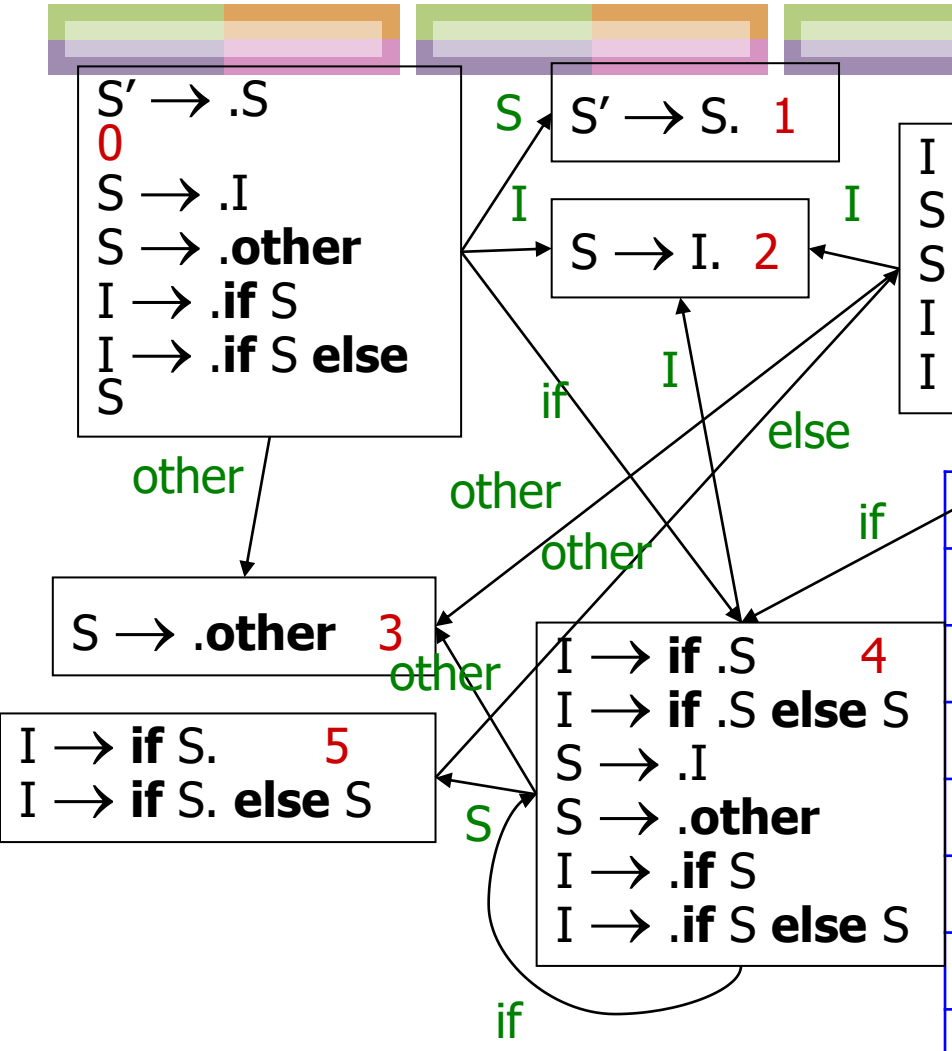
$S \rightarrow (S)S \mid \lambda$

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				ACC		
2		R1		R1		
3		R2		R2		
4	S4		S3		5	2
5		S6		R3		
6	S4		S3		7	2
7		R4		R4		

End