

Top-down Parsing

CMPT 379: Compilers

Instructor: Anoop Sarkar

anoopsarkar.github.io/compilers-class

Parsing - Roadmap

- Parser:
 - decision procedure: builds a parse tree
- Top-down vs. bottom-up
- LL(1) – Deterministic Parsing
 - recursive-descent
 - table-driven
- LR(k) – Deterministic Parsing
 - LR(0), SLR(1), LR(1), LALR(1)
- Parsing arbitrary CFGs – Polynomial time parsing

Top-Down vs. Bottom Up

Grammar: $S \rightarrow A B$

Input String: ccbca

$A \rightarrow c \mid \epsilon$

$B \rightarrow cbB \mid ca$

Top-Down/leftmost		Bottom-Up/rightmost	
$S \Rightarrow AB$	$S \rightarrow AB$	$ccbca \Leftarrow Acbca$	$A \rightarrow c$
$\Rightarrow cB$	$A \rightarrow c$	$\Leftarrow AcbB$	$B \rightarrow ca$
$\Rightarrow ccbB$	$B \rightarrow cbB$	$\Leftarrow AB$	$B \rightarrow cbB$
$\Rightarrow ccbca$	$B \rightarrow ca$	$\Leftarrow S$	$S \rightarrow AB$

Leftmost derivation for **id + id * id**

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow - E$

$E \rightarrow \text{id}$

$E \Rightarrow E + E$

$\Rightarrow \text{id} + E$

$\Rightarrow \text{id} + E * E$

$\Rightarrow \text{id} + \text{id} * E$

$\Rightarrow \text{id} + \text{id} * \text{id}$

$E \Rightarrow_{\text{lm}}^* \text{id} + E * E$

Predictive Top-Down Parser

- Knows which production to choose based on single lookahead symbol
- Need LL(1) grammars
 - First L: reads input Left to right
 - Second L: produce Leftmost derivation
 - 1: one symbol of lookahead
- Cannot have left-recursion
- Must be left-factored (no left-factors)
- Not all grammars can be made LL(1)

LL(1) Parser

- In recursive-descent
 - for each non-terminal and input token, many choices of production to use
 - Backtracking to remove bad choices
- In LL(1)
 - for each non-terminal and each token, only one production

$S \rightarrow^* \omega A \beta$ and next input token: t

$A \rightarrow \alpha$ is the only production

$\omega \alpha \beta$

Left Factoring

- Consider this grammar
 - $E \rightarrow T + E \mid T$
 - $T \rightarrow id \mid id * T \mid (E)$
- Hard to predict because
 - For T two productions start with id
 - For E it is not clear how to predict
- The grammar must not have left-recursion
- The grammar should be left-factored

Left Factoring

- In general, for rules

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$$

- Left factoring is achieved by the following grammar transformation:

$$A \rightarrow \alpha A' \mid \gamma$$

$$A' \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

Left Factoring

- Recall the grammar
 - $E \rightarrow T + E \mid T$
 - $T \rightarrow \text{id} \mid \text{id} * T \mid (E)$
- Factor out common prefixes for productions
 - $E \rightarrow T X$
 - $X \rightarrow + E \mid \varepsilon$
 - $T \rightarrow \text{id} Y \mid (E)$
 - $Y \rightarrow * T \mid \varepsilon$

Predictive Parsing Table

- Can be specified via 2D tables
 - One dimension for current (leftmost) non-terminal to expand
 - One dimension for next token
 - Each table entry contains one production

Productions	
1	$E \rightarrow T X$
2	$X \rightarrow \epsilon$
3	$X \rightarrow + E$
4	$T \rightarrow (E)$
5	$T \rightarrow \text{id } Y$
6	$Y \rightarrow * T$
7	$Y \rightarrow \epsilon$

	+	*	()	id	\$
E			$T X$		$T X$	
X	$+ E$			ϵ		ϵ
T			(E)		$\text{id } Y$	
Y	ϵ	$* T$		ϵ		ϵ

Predictive Parsing Table

- Consider $[E, id]$ entry
 - When current non-terminal is E and the next input is id , use production $E \rightarrow T X$

Productions	
1	$E \rightarrow T X$
2	$X \rightarrow \epsilon$
3	$X \rightarrow + E$
4	$T \rightarrow (E)$
5	$T \rightarrow id Y$
6	$Y \rightarrow * T$
7	$Y \rightarrow \epsilon$

	+	*	()	id	\$
E			$T X$		$T X$	
X	$+ E$			ϵ		ϵ
T			(E)		$id Y$	
Y	ϵ	$* T$		ϵ		ϵ

Predictive Parsing Table

- Consider $[Y, +]$ entry
 - When current non-terminal is Y and the next input is $+$, get rid of Y
 - Y can be followed by $+$ only if $Y \rightarrow \epsilon$

Productions	
1	$E \rightarrow T X$
2	$X \rightarrow \epsilon$
3	$X \rightarrow + E$
4	$T \rightarrow (E)$
5	$T \rightarrow \text{id } Y$
6	$Y \rightarrow * T$
7	$Y \rightarrow \epsilon$

	$+$	$*$	$($	$)$	id	$\$$
E			$T X$		$T X$	
X	$+ E$			ϵ		ϵ
T			(E)		$\text{id } Y$	
Y	ϵ	$* T$		ϵ		ϵ

Predictive Parsing Table

- Blank entries indicate error situations
- Consider $[E, *]$ entry
 - There is no way to derive a string starting with $*$ from non-terminal E

Productions	
1	$E \rightarrow T X$
2	$X \rightarrow \epsilon$
3	$X \rightarrow + E$
4	$T \rightarrow (E)$
5	$T \rightarrow \text{id } Y$
6	$Y \rightarrow * T$
7	$Y \rightarrow \epsilon$

	+	*	()	id	\$
E			$T X$		$T X$	
X	$+ E$			ϵ		ϵ
T			(E)		$\text{id } Y$	
Y	ϵ	$* T$		ϵ		ϵ

Predictive Parsing

- Method similar to recursive descent, except
 - For each non-terminal S
 - We look at the next token a
 - And chose the production shown at entry $[S,a]$
- We use a stack to keep track of pending non-terminals (frontier of parse tree)
- We reject when we encounter an error state
- We accept when we encounter end-of-input and empty stack

Table-Driven Parsing

```
stack.push($); stack.push(S);  
a = input.read();
```

```
forever do begin
```

```
    X = stack.peek();
```

```
    if X = a and a = $ then return SUCCESS;
```

```
    elseif X = a and a != $ then
```

```
        stack.pop(X); a = input.read();
```

```
    elseif X != a and  $X \in N$  and  $M[X, a]$  not empty then
```

```
        stack.pop(X);
```

```
        stack.push( $M[X, a]$ ); /*  $M[X, a] = Y_1 \dots Y_n$  */  
        else ERROR!
```

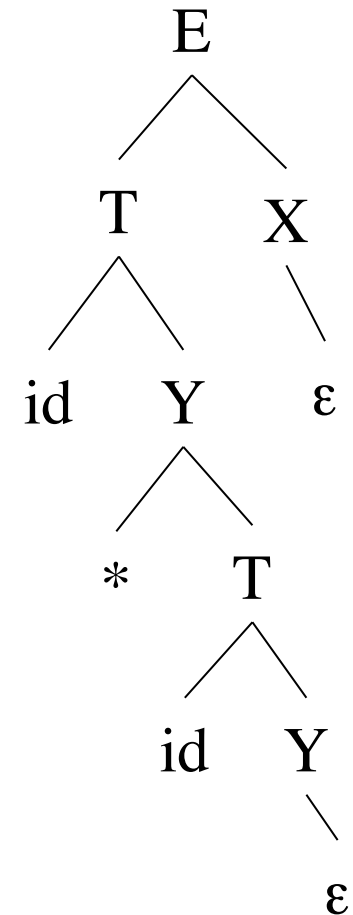
```
end
```

**Stack: to keep track
of pending non-terminals
in the derivation
(leaves in parse tree)**

Trace “id*id”

	+	*	()	id	\$
E			T X		T X	
X	+ E			ϵ		ϵ
T			(E)		id Y	
Y	ϵ	* T		ϵ		ϵ

Stack	Input	Action
E \$	id*id\$	T X
T X \$	id*id\$	id Y
id Y X \$	id*id\$	terminal
Y X \$	*id\$	* T
* T X \$	*id\$	terminal
T X \$	id\$	id Y
id Y X \$	id\$	terminal
Y X \$	\$	ϵ
X \$	\$	ϵ
\$	\$	Accept!



When to pick $Y \rightarrow \epsilon$?

Productions	
1	$E \rightarrow T X$
2	$X \rightarrow \epsilon$
3	$X \rightarrow + E$
4	$T \rightarrow (E)$
5	$T \rightarrow \text{id } Y$
6	$Y \rightarrow * T$
7	$Y \rightarrow \epsilon$

- Choice between $Y \rightarrow * T$ and $Y \rightarrow \epsilon$
- $\text{FIRST}(*T) = \{ * \}$
- For $Y \rightarrow \epsilon$ we compute $\text{FOLLOW}(Y)$
- $\text{FOLLOW}(Y) = ?$
- $\text{FOLLOW}(Y) = \text{FOLLOW}(T)$
- $\text{FOLLOW}(T) = (\text{FIRST}(X) - \{ \epsilon \}) + \text{FOLLOW}(E)$
- $\text{FOLLOW}(T) = \{ + ,) , \$ \}$
- $\text{FOLLOW}(Y) = \{ + ,) , \$ \}$

Predictive Parsing table

- Given a grammar produce the predictive parsing table
- We need to know for all rules $A \rightarrow \alpha \mid \beta$ the lookahead symbol
- Based on the lookahead symbol the table can be used to pick which rule to push onto the stack
- This can be done using two sets: FIRST and FOLLOW

Predictive Parsing Table

- For Nonterminal A , rule $A \rightarrow \alpha$, and the token t , $M[A, t] = \alpha$ in two cases:
- If $\alpha \Rightarrow^* t \beta$
 - α can derive a t in the first position
 - We say that $t \in \text{First}(\alpha)$
- $A \rightarrow \alpha$ and $\alpha \Rightarrow^* \varepsilon$ and $S \Rightarrow^* \beta A t \delta$
 - Useful if stack has A , input is t and A cannot derive t
 - In this case only option is to get rid of A (by $\alpha \Rightarrow^* \varepsilon$)
 - Can work only if t can follow A in at least on derivation
 - We say $t \in \text{Follow}(A)$

FIRST and FOLLOW

$a \in \text{FIRST}(\alpha)$ if $\alpha \Rightarrow^* a\beta$

if $\alpha \Rightarrow^* \epsilon$ then $\epsilon \in \text{FIRST}(\alpha)$

$a \in \text{FOLLOW}(A)$ if $S \Rightarrow^* \alpha A a \beta$

$a \in \text{FOLLOW}(A)$ if $S \Rightarrow^* \alpha A \gamma a \beta$

and $\gamma \Rightarrow^* \epsilon$

Conditions for LL(1)

- Necessary conditions:
 - no ambiguity
 - no left recursion
 - Left factored grammar
- A grammar G is LL(1) if - whenever
 $A \rightarrow \alpha \mid \beta$
 1. $\text{First}(\alpha) \cap \text{First}(\beta) = \emptyset$
 2. $\alpha \Rightarrow^* \epsilon$ implies $\neg(\beta \Rightarrow^* \epsilon)$
 3. $\alpha \Rightarrow^* \epsilon$ implies $\text{First}(\beta) \cap \text{Follow}(A) = \emptyset$

ComputeFirst(α : string of symbols)

```
// assume  $\alpha = X_1 X_2 X_3 \dots X_n$   
if  $X_1 \in T$  then First[ $\alpha$ ] := { $X_1$ }  
else begin  
   $i := 1$ ; First[ $\alpha$ ] := ComputeFirst( $X_1$ ) \ { $\epsilon$ };  
  while  $X_i \Rightarrow^* \epsilon$  do begin  
    if  $i < n$  then  
      First[ $\alpha$ ] := First[ $\alpha$ ]  $\cup$  ComputeFirst( $X_{i+1}$ ) \ { $\epsilon$ };  
    else  
      First[ $\alpha$ ] := First[ $\alpha$ ]  $\cup$  { $\epsilon$ };  
     $i := i + 1$ ;  
  end  
end
```

Recursion in computing FIRST
causes problems when faced with
recursive grammar rules

ComputeFirst; modified

```
foreach  $X \in T$  do First[X] := {X};  
foreach  $p \in P : X \rightarrow \epsilon$  do First[X] := { $\epsilon$ };  
repeat foreach  $X \in N, p : X \rightarrow Y_1 Y_2 Y_3 \dots Y_n$  do  
  begin  $i := 1$ ;  
    while  $Y_i \Rightarrow^* \epsilon$  and  $i \leq n$  do begin  
      First[X] := First[X]  $\cup$  First[ $Y_i$ ] \ { $\epsilon$ };  
       $i := i + 1$ ;  
    end  
    if  $i = n + 1$  then First[X] := First[X]  $\cup$  { $\epsilon$ };  
until no change in First[X] for any X;
```

ComputeFirst; modified

```
foreach  $X \in T$  do First[X] := X;  
foreach  $p \in P : X \rightarrow \epsilon$  do First[X] := { $\epsilon$ };  
repeat foreach  $X \in N, p : X \rightarrow Y_1 Y_2 Y_3 \dots Y_n$  do  
  begin  $i := 1$ ;  
    while  $Y_i \Rightarrow^*$  do  
      First[X] := First[X]  $\cup$  First[Yi];  
       $i := i + 1$ ;  
    end  
  if  $i = n + 1$  then First[X] := First[X]  $\cup$  { $\epsilon$ };  
until no change in First[X] for any X;
```

Non-recursive FIRST computation works with left-recursive grammars.
Computes a fixed point for FIRST[X] for all non-terminals X in the grammar.
But this algorithm is very inefficient.

First Sets

$\text{First}(+) = \{+\}$

$\text{First}(\ast) = \{\ast\}$

$\text{First}(\text{'('}) = \{\text{'('}\}$

$\text{First}(\text{'('}) = \{\text{'('}\}$

$\text{First}(\text{id}) = \{\text{id}\}$

$\text{First}(E) = ?$

$\text{First}(T) \subseteq \text{First}(E)$

$\text{First}(T) = \{\text{id}, \text{'('}\}$

$\text{First}(E) = \{\text{id}, \text{'('}\}$

$\text{First}(X) = \{+, \epsilon\}$

$\text{First}(Y) = \{\ast, \epsilon\}$

Productions	
1	$E \rightarrow T X$
2	$X \rightarrow \epsilon$
3	$X \rightarrow + E$
4	$T \rightarrow (E)$
5	$T \rightarrow \text{id } Y$
6	$Y \rightarrow \ast T$
7	$Y \rightarrow \epsilon$

Follow Sets

- Algorithm sketch
 1. Add $\$$ to $\text{Follow}(S)$
 2. For each production $A \rightarrow \alpha X \beta$
 - Add $\text{First}(\beta) - \{\epsilon\}$ to $\text{Follow}(X)$
 3. For each $A \rightarrow \alpha X \beta$ where $\epsilon \in \text{First}(\beta)$
 - Add $\text{Follow}(A)$ to $\text{Follow}(X)$
- Repeat steps 2-3 until no follow set grows

ComputeFollow

```
Follow(S) := {$};  
repeat  
  foreach  $p \in P$  do  
    case  $p = A \rightarrow \alpha B \beta$  begin  
      Follow[B] := Follow[B]  $\cup$  ComputeFirst( $\beta$ ) \  $\{\epsilon\}$ ;  
      if  $\epsilon \in \text{First}(\beta)$  then  
        Follow[B] := Follow[B]  $\cup$  Follow[A];  
      end  
    case  $p = A \rightarrow \alpha B$   
      Follow[B] := Follow[B]  $\cup$  Follow[A];  
until no change in any Follow[N]
```

Follow Sets. Example

$\text{Follow}(E) \subseteq \text{Follow}(X)$
 $\text{Follow}(X) \subseteq \text{Follow}(E)$
 $\text{First}(X) - \{\epsilon\} \subseteq \text{Follow}(T)$
 $\text{Follow}(E) \subseteq \text{Follow}(T)$
 $\text{Follow}(Y) \subseteq \text{Follow}(T)$
 $\text{Follow}(T) \subseteq \text{Follow}(Y)$

$\text{Follow}(E) = \{\$,)\}$
 $\text{Follow}(X) = \{\$,)\}$
 $\text{Follow}(T) = \{+, \$,)\}$
 $\text{Follow}(Y) = \{+, \$,)\}$
 $\text{Follow}('(') = \{(\, \text{id}\}$
 $\text{Follow}(')') = \{+, \$,)\}$
 $\text{Follow}(+) = \{(\, \text{id}\}$
 $\text{Follow}(*) = \{(\, \text{id}\}$
 $\text{Follow}(\text{id}) = \{*, +, \$,)\}$

Productions	
1	$E \rightarrow T X$
2	$X \rightarrow \epsilon$
3	$X \rightarrow + E$
4	$T \rightarrow (E)$
5	$T \rightarrow \text{id } Y$
6	$Y \rightarrow * T$
7	$Y \rightarrow \epsilon$

Building the Parse Table

- Compute First and Follow sets
- For each production $A \rightarrow \alpha$
 - For each $t \in \text{First}(\alpha)$
 - $M[A, t] = \alpha$
 - If $\epsilon \in \text{First}(\alpha)$, for each $t \in \text{Follow}(A)$
 - $M[A, t] = \alpha$
 - If $\epsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(\alpha)$
 - $M[A, \$] = \alpha$
 - All undefined entries are errors

Predictive Parsing Table

$\text{First}(E) = \{\text{id}, '('\}$
 $\text{Follow}(E) = \{\$, '\}\}$
 $\text{First}(X) = \{+, \epsilon\}$
 $\text{Follow}(X) = \{\$, '\}\}$

$\text{First}(T) = \{\text{id}, '('\}$
 $\text{Follow}(T) = \{+, \$, '\}\}$
 $\text{First}(Y) = \{*, \epsilon\}$
 $\text{Follow}(Y) = \{+, \$, '\}\}$

Productions	
1	$E \rightarrow T X$
2	$X \rightarrow \epsilon$
3	$X \rightarrow + E$
4	$T \rightarrow (E)$
5	$T \rightarrow \text{id } Y$
6	$Y \rightarrow * T$
7	$Y \rightarrow \epsilon$

	+	*	()	id	\$
E			T X		T X	
X	+ E			ϵ		ϵ
T			(E)		id Y	
Y	ϵ	* T		ϵ		ϵ

Example First/Follow

$$S \rightarrow AB$$

$$A \rightarrow c \mid \varepsilon$$

Not an LL(1) grammar

$$B \rightarrow cbB \mid ca$$

$$\text{First}(A) = \{c, \varepsilon\}$$

$$\text{Follow}(A) = \{c\}$$

$$\text{First}(B) = \{c\}$$

$$\text{Follow}(A) \cap$$

$$\text{First}(cbB) =$$

$$\text{First}(c) = \{c\}$$

$$\text{First}(ca) = \{c\}$$

$$\text{Follow}(B) = \{\$ \}$$

$$\text{First}(S) = \{c\}$$

$$\text{Follow}(S) = \{\$ \}$$

Converting to LL(1)

$$S \rightarrow AB$$

$$A \rightarrow c \mid \varepsilon$$

$$B \rightarrow cbB \mid ca$$

Note that grammar
is regular: $c? (cb)^* ca$

$c (c b c b \dots c b) c a$
 $(c b c b \dots c b) c a$



$c c (b c b \dots c b c) a$
 $c (b c b \dots c b c) a$

same as:

$c c? (bc)^* a$

$$S \rightarrow cAa$$

$$A \rightarrow cB \mid B$$

$$B \rightarrow bcB \mid \varepsilon$$

Verifying LL(1) using F/F sets

$$S \rightarrow cAa$$

$$A \rightarrow cB \mid B$$

$$B \rightarrow bcB \mid \varepsilon$$

$$\text{First}(A) = \{b, c, \varepsilon\}$$

$$\text{Follow}(A) = \{a\}$$

$$\text{First}(B) = \{b, \varepsilon\}$$

$$\text{Follow}(B) = \{a\}$$

$$\text{First}(S) = \{c\}$$

$$\text{Follow}(S) = \{\$ \}$$

Building the Parse Table

- Compute First and Follow sets
- For each production $A \rightarrow \alpha$
 - foreach $a \in \text{First}(\alpha)$ add $A \rightarrow \alpha$ to $M[A,a]$
 - If $\epsilon \in \text{First}(\alpha)$ add $A \rightarrow \alpha$ to $M[A,b]$ for each b in $\text{Follow}(A)$
 - If $\epsilon \in \text{First}(\alpha)$ add $A \rightarrow \alpha$ to $M[A,\$]$ if $\$ \in \text{Follow}(\alpha)$
 - All undefined entries are errors

Predictive Parsing Table

Productions	
1	$T \rightarrow F T'$
2	$T' \rightarrow \epsilon$
3	$T' \rightarrow * F T'$
4	$F \rightarrow \text{id}$
5	$F \rightarrow (T)$

$\text{FIRST}(T) = \{\text{id}, (\}$
 $\text{FIRST}(T') = \{*, \epsilon\}$
 $\text{FIRST}(F) = \{\text{id}, (\}$

$\text{FOLLOW}(T) = \{\$,)\}$
 $\text{FOLLOW}(T') = \{\$,)\}$
 $\text{FOLLOW}(F) = \{*, \$,)\}$

	*	()	id	\$
T		$T \rightarrow F T'$		$T \rightarrow F T'$	
T'	$T' \rightarrow * F T'$		$T' \rightarrow \epsilon$		$T' \rightarrow \epsilon$
F		$F \rightarrow (T)$		$F \rightarrow \text{id}$	

Revisit conditions for LL(1)

- A grammar G is LL(1) iff - whenever $A \rightarrow \alpha \mid \beta$
 1. $\text{First}(\alpha) \cap \text{First}(\beta) = \emptyset$
 2. $\alpha \Rightarrow^* \varepsilon$ implies $\neg(\beta \Rightarrow^* \varepsilon)$
 3. $\alpha \Rightarrow^* \varepsilon$ implies $\text{First}(\beta) \cap \text{Follow}(A) = \emptyset$
- No more than one entry per table field

Error Handling

- Reporting & Recovery
 - Report as soon as possible
 - Suitable error messages
 - Resume after error
 - Avoid cascading errors
- Phrase-level vs. Panic-mode recovery

Panic-Mode Recovery

- Skip tokens until *synchronizing set* is seen
 - Follow(A)
 - garbage or missing things after
 - Higher-level start symbols
 - First(A)
 - garbage before
 - Epsilon
 - if nullable
 - Pop/Insert terminal
 - “auto-insert”
- Add “synch” actions to table

Summary so far

- LL(1) grammars, necessary conditions
 - No left recursion
 - Left-factored
- Not all languages can be generated by LL(1) grammar
- LL(1) – Parsing: $O(n)$ time complexity
 - recursive-descent and table-driven predictive parsing
- LL(1) grammars can be parsed by simple predictive recursive-descent parser
 - Alternative: table-driven top-down parser

Extra Slides

ComputeFirst on Left-recursive Grammars

- ComputeFirst as defined earlier loops on left-recursive grammars
- Here is an alternative algorithm for ComputeFirst
 1. Compute non left-recursive cases of FIRST
 2. Create a graph of recursive cases where FIRST of a non-terminal depends on another non-terminal
 3. Compute Strongly Connected Components (SCC)
 4. Compute FIRST starting from root of SCC to avoid cycles

ComputeFirst on Left-recursive Grammars

- Each Strongly Connected Component can have recursion
- But the connections between SCC means that (by defn) what we have now is a directed acyclic graph – hence without left recursion
- Unlike top-down LL parsing, bottom-up LR parsing allows left-recursive grammars, so this algorithm is useful for LR parsing

ComputeFirst on Left-recursive Grammars

- $S \rightarrow BD \mid D$
- $D \rightarrow d \mid Sd$

$\text{FIRST}_0[A] := \{a\}$

$\text{FIRST}_0[C] := \{\}$

$\text{FIRST}_0[B] := \{b\}$

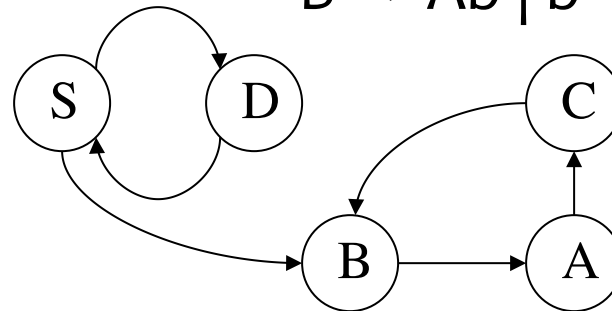
$\text{FIRST}_0[S] := \{b, d\}$

$\text{FIRST}_0[D] := \{d\}$

- $A \rightarrow CB \mid a$

- $C \rightarrow Bb \mid \epsilon$

- $B \rightarrow Ab \mid b$



Compute
Strongly
Connected
Components

2 SCCs: e.g. consider B-A-C

$\text{FIRST}[B] := \text{FIRST}_0[B] + \text{ComputeFirst}(A)$

$\text{FIRST}[A] := \text{FIRST}_0[A] + \text{ComputeFirst}(C)$

$\text{FIRST}[A] := \text{FIRST}[A] + \text{FIRST}_0[B]$

$\text{FIRST}[C] := \text{FIRST}_0[C] + \text{FIRST}_0[B]$

$\text{FIRST}[C] := \text{FIRST}[C] + \{\epsilon\}$

Examples

$S \rightarrow A B C$

$A \rightarrow a \mid \varepsilon$

$B \rightarrow b B \mid \varepsilon$

$C \rightarrow c \mid \varepsilon$

Is this LL(1)?

$S \rightarrow F$

$F \rightarrow A (B) \mid B A$

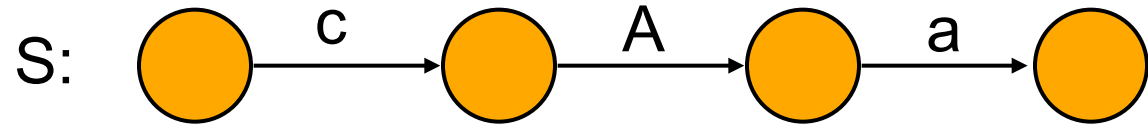
$A \rightarrow x \mid y$

$B \rightarrow a B \mid b B \mid \varepsilon$

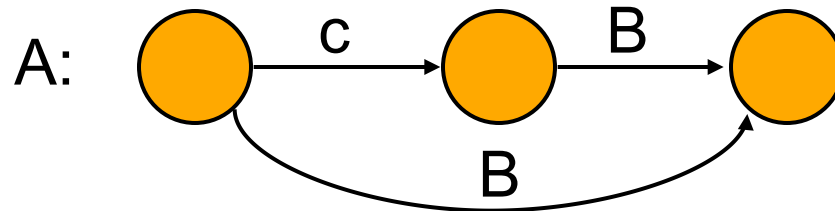
Is this LL(1)?

Transition Diagram

$S \rightarrow cAa$



$A \rightarrow cB \mid B$



$B \rightarrow bcB \mid \epsilon$

