Chapter 4
Top-Down Parsing

한양대학교 컴퓨터공학부

컴파일러

2014년 2학기

Top-down parsing

- Top-down parsing
  - Definition
    - Parsing an input string of tokens by tracing out the steps in a **leftmost derivation**.
  - Categories
    - Backtracking parsers
      - Powerful but slow
    - Predictive parsers
      - Using one or more lookahead tokens
      - Recursive-descent parsing
      - LL(1) parsing.

Recursive-descent parsing

a grammer should always be translated into EBNF if recursive-descent is to be used

- Recursive descent parsing
  - Each procedure is generated for each grammar rule.
- Expression grammar
  - *exp* → *exp addop term* | *term*
  - *addop* → + / −
  - *term* → *term mulop factor* / *factor*
  - *mulop* → *
  - *factor* → (*exp*) / ***number***

5 procedures are required.

Recursive-descent parsing

- *factor* → (*exp*) | ***number***

- *terminal: match*()
- *nonterminal: function*

```
procedure factor
begin
   case token of
   (:   match( ( ) ;
        exp ;
        match( ) ) ;
    number :
        match(number) ;
    else error ;
    end case ;
end factor ;
```

Recursive-descent parsing

```
procedure match( expectedToken) ;
begin
  if token = expectedToken then
    getToken ;
  else
    error ;
  end if ;
end match ;
```

- *compare match*(**(**) *with match*(**)**) in **procedure** *factor*.

Recursive-descent parsing

- *if-stmt* → **if (** *exp* **)** *statement*
  | **if (** *exp* **)** *statement* **else** *statement*
- **EBNF**
  - *if-stmt* → **if (** *exp* **)** *statement* [**else** *statement*]

```
procedure ifStmt ;
begin
  match ( if ) ;
  match ( ( ) ;
  exp ;
  match ( ) ) ;
  statement ;
  if token = else then
    match ( else ) ;
    statement ;
  end if ;
end ifStmt ;
```

Recursive-descent parsing

- *exp* → *exp addop term* | *term*

BNF를 ENBF로 변경하지 않고 parse처리시 left recursive grammer에서 문제가 발생한다. (ambiguity 발생 exp-> exp addop term | term에서 token이 exp인지 term인지 알 수 없다.)

procedure *exp* ;
begin
case *token* **of**
  ?: *exp* ;
    addop ;
    term;
  ?: *term*;
end case
end *exp* **;**

Recursive-descent parsing

- *exp* → *exp addop term* | *term*
- ***EBNF***
  - *exp* → *term* { *addop term* }

```
procedure exp ;
begin
  term ;
  while token = + or token = - do
    match (token) ;
    term ;
  end while ;
end exp ;
```

Recursive-descent parsing

- *term* → *term mulop factor* | *factor*
- ***EBNF***
  - *term* → *factor* { *mulop factor* }

```
procedure term ;
begin
  factor ;
  while token = * do
    match (token) ;
    factor ;
  end while ;
end term ;
```

Recursive-descent parsing

- *Left associativity is conserved.*
  - A simple integer arithmetic
    - *exp* → *term* { *addop term* }

```
function exp : integer ;
var temp : integer ;
begin
  temp := term ;
  while token = + or token = - do
    case token of
     + : match ( + ) ;
         temp := temp + term ;
     - : match ( - ) ;
         temp := temp – term ;
     end case ;
  end while ;
  return temp ;
end exp ;
```

- A simple calculator
  - p. 148-149

Recursive-descent parsing

- Syntax tree generation

```
function exp : syntaxTree ;
var temp, newtemp : syntaxTree ;
begin
  temp := term ;
  while token = + or token = - do
    case token of
     + : match (+) ;
         newtemp : = makeOpNode( + ) ;
         leftChild(newtemp) := temp ;
         rightChild(newtemp) := term ;
         temp := newtemp ;
     - : match (-) ;
         newtemp := makeOpNode( - ) ;
         leftChild(newtemp) := temp ;
         rightChild(newtemp) := term ;
         temp := newtemp ;
    end case ;
  end while ;
  return temp ;
end exp ;
```

Recursive-descent parsing

- Syntax tree generation

```
function exp : syntaxTree ;
var temp, newtemp : syntaxTree ;
begin
  temp := term ;
  while token = + or token = - do
    newtemp : = makeOpNode(token) ;
    match (token) ;
    leftChild(newtemp) := temp ;
    rightChild(newtemp) := term ;
    temp := newtemp ;
  end while ;
  return temp ;
end exp ;
```

Recursive-descent parsing

- Syntax tree generation

```
function ifStatement : syntaxTree
var temp : syntaxTree ;
begin
  match( if ) ;
  match( ( ) ;
  temp := makeStmtNode( if ) ;
  testChild(temp) := exp ;
  match ( ) ) ;
  thenChild(temp) := statement ;
  if token = else then
    match(else) ;
    elseChild(temp) := statement ;
  else
    elseChild(temp) := nil ;
  end if ;
end ifStatement ;
```

Recursive-descent parsing

- **Difficulties**
  - BNF → EBNF is not easy.
  - A → α | β …
    - The first sets should be determined.

      backtracking이 존재하지 않는 predictive parser이기 때문에, ambiguity 처리 불가능

LL(1) Parsing

- **gets its name as follows**
  - **“L” : process input from left to right**
  - **“L” : leftmost derivation**
  - **“1” : only one symbol for lookahead**
- **The basic example of LL(1) parsing**
  - grammar
    - $S \rightarrow (S)\ S \mid \varepsilon$
  - Input string
    - ( )

$$
\begin{aligned}
S &\Rightarrow (S)S \\
&\Rightarrow (\ )S \\
&\Rightarrow (\ )
\end{aligned}
$$

LL(1) Parsing

- **The basic example of LL(1) parsing**
  - grammar
    - $S \rightarrow (S)\ S \mid \varepsilon$
  - Input string
    - ( )

input left부터 처리 (input이 left일수록 top에 위치)

derivation을 left부터 처리
(grammer의 left일수록 top에 위치)

Parsing stack	Input	Action
$ S	( ) $	$S \rightarrow (\ S\ )\ S$
$ S) S (	( ) $	match
$ S) S	) $	$S \rightarrow \varepsilon$
$ S)	) $	match
$ S	$	$S \rightarrow \varepsilon$
$	$	accept

$ - bottom of stack

- $S \Rightarrow (S)S$
  $\Rightarrow (\ )S$
  $\Rightarrow (\ )$

LL(1) Parsing

- **Outline**
  - **Initialization**
    - Put the start symbol in the stack
  - **Iteration of the followings until the stack is empty.**
    - If a **nonterminal** is at the stack top,
      - replace the nonterminal using a grammar rule.
    - If a **token** is at the stack top,
      - match.
  - **If the stack is empty,**
    - if the input string is empty, accept.
    - otherwise, reject.

LL(1) Parsing Table

- **LL(1) parsing table**
  - A two-dimensional array indexed by nonterminals and terminals.
  - It contains production choices to use at the appropriate parsing step.

$M[N, T]$	(	)	$
S	$S \rightarrow (S) S$	$S \rightarrow \varepsilon$	$S \rightarrow \varepsilon$

- Once a parsing table is given, LL(1) parsing is simple.
  - Figure 4.2

LL(1) Parsing Table

- **LL(1) parsing table generation**
  - A table entry $M[A, \boldsymbol{a}]$ has every grammar rule $A \rightarrow \alpha$
    - if there is a derivation $\alpha \Rightarrow^* \boldsymbol{a}\beta$ or
    - if there is a derivation $\alpha \Rightarrow^* \boldsymbol{\varepsilon}$ and $S \Rightarrow^* \beta A \boldsymbol{a} \gamma$ for start symbol S.

***M*[*N*, *T*]**	**(**	**)**	**$**
S	$S \rightarrow (S) S$	$S \rightarrow \varepsilon$	$S \rightarrow \varepsilon$

LL(1) Grammar

- **LL(1) grammar**
  - The LL(1) parsing table has **at most one production in each entry**.
- **An LL(1) grammar cannot be ambiguous.**

$M[N, T]$	(	)	\$
S	$S \rightarrow (S) S$	$S \rightarrow \varepsilon$	$S \rightarrow \varepsilon$

Disambiguating rule

statement → *if-stmt* | ***other***

if-stmt → **if (** *exp* **)** *statement else-part*

else-part → **else** *statement* | ε

exp → **0** | **1**

***M*[*N*,*T*]**	**if**	***other***	**else**	**0**	**1**	**$**
statement	*statement* → *if-stmt*	*statement* → ***other***				
if-stmt	*if-stmt* → **if (** *exp* **)** *statement else-part*					
else-part			*else-part* → **else** *statement*			*else-part* → ε
exp				*exp*→0	*exp*→1	

Parsing for if (0) if (1) *other* else *other*

Parsing stack	Input	Action
$ *S*	**i (0) i (1) o e o $**	$S \rightarrow I$
$ *I*	**i (0) i (1) o e o $**	$I \rightarrow$ **i (** E **)** $S\ L$
$ *L S* **)** *E* **(i**	**i (0) i (1) o e o $**	match
$ *L S* **)** *E* **(**	**(0) i (1) o e o $**	match
$ *L S* **)** *E*	**0) i (1) o e o $**	$E \rightarrow$ **0**
$ *L S* **) 0**	**0) i (1) o e o $**	match
$ *L S* **)**	**) i (1) o e o $**	match
$ *L S*	**i (1) o e o $**	$S \rightarrow I$
$ *L I*	**i (1) o e o $**	$I \rightarrow$ **i (** E **)** $S\ L$
$ *L L S* **)** *E* **(i**	**i (1) o e o $**	match

S = statement, *I = if-stmt*, *L=else-part*, *E=exp*, **i=if**, **e=else**, **o=*other***.

Parsing for if (0) if (1) *other* else *other*

Parsing stack	Input	Action
$\$\ L\ L\ S\ \mathbf{)}\ E\ \mathbf{(}\ \mathbf{i}$	**i (1) o e o $**	match
$\$\ L\ L\ S\ \mathbf{)}\ E\ \mathbf{(}$	**(1) o e o $**	match
$\$\ L\ L\ S\ \mathbf{)}\ E$	**1) o e o $**	$E \rightarrow \mathbf{1}$
$\$\ L\ L\ S\ \mathbf{)}\ \mathbf{1}$	**1) o e o $**	match
$\$\ L\ L\ S\ \mathbf{)}$	**) o e o $**	match
$\$\ L\ L\ S$	**o e o $**	$S \rightarrow \mathbf{o}$
$\$\ L\ L\ \mathbf{o}$	**o e o $**	match
$\$\ L\ L$	**e o $**	$L \rightarrow \mathbf{e}\ S$
$\$\ L\ S\ \mathbf{e}$	**e o $**	match
$\$\ L\ S$	**o $**	$S \rightarrow \mathbf{o}$
$\$\ L\ \mathbf{o}$	**o $**	match
$\$\ L$	**$**	$L \rightarrow \boldsymbol{\varepsilon}$
$\$$	**$**	accept

Left Recursion Removal

- **Left recusion**
  - **Immediate left recursion**
    - *exp* → *exp addop term* | *term*
    - *exp* → *exp* + *term* | *exp* - *term* | *term*
  - **Indirect left recursion**
    - *A* → *Bb*| …
    - *B* → *Aa*| …

Left Recursion Removal

- **Simple immediate left recusion removal**

$$A \rightarrow A\alpha \mid \beta \quad \xRightarrow{\beta\alpha^*} \quad \begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \varepsilon \end{array}$$

- **Example 4.1**
  - *exp* → *exp addop term* | *term*
    - $A = exp$
    - $\alpha = addop\ term$
    - $\beta = term$

$$\begin{array}{l} exp \rightarrow term\ exp' \\ exp' \rightarrow addop\ term\ exp' \mid \varepsilon \end{array}$$

Left Recursion Removal

- **General immediate left recusion removal**

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$

$$\Downarrow \quad (\beta_1 \mid \beta_2 \mid \ldots \mid \beta_m)(\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n)^*$$

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon$$

- **Example 4.2**
  - $exp \rightarrow exp + term \mid exp - term \mid term$
    - $A = exp$, $\alpha_1 = + term$, $\alpha_2 = - term$, $\beta = term$

$$exp \rightarrow term\ exp'$$

$$exp' \rightarrow + term\ exp' \mid - term\ exp' \mid \varepsilon$$

Left Recursion Removal

- **General left recursion removal (skip)**

for i := 1 **to** m **do**
  for j := 1 **to** i - 1 **do**
    replace each grammar rule choice of the form $A_i \rightarrow A_j \beta$ *by the rule*
      $A_i \rightarrow \alpha_1\beta \mid \alpha_2\beta \mid \ldots \mid \alpha_k\beta$, *where* $A_j \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$ is
      the current rule for A_j

- **Example 4.3**

$$A \rightarrow Ba \mid Aa \mid c$$
$$B \rightarrow Bb \mid Ab \mid d$$

$\Longrightarrow$

$$A \rightarrow BaA' \mid cA'$$
$$A' \rightarrow aA' \mid \varepsilon$$
$$B \rightarrow cA'bB' \mid dB'$$
$$B' \rightarrow bB' \mid aA'bB' \mid \varepsilon$$

Left Recursion Removal

- Simple arithmetic expression grammar with left recursion removed.

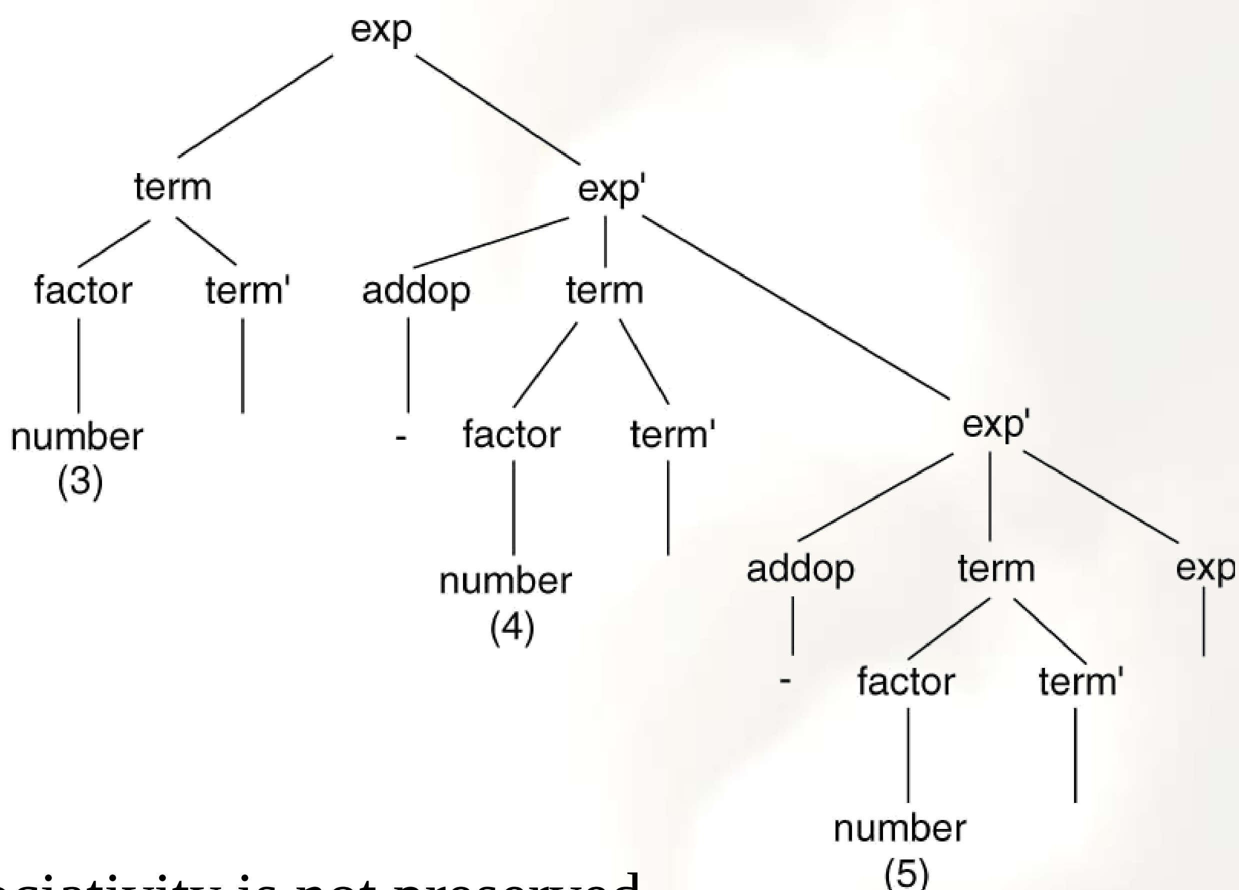
$$
\begin{aligned}
&exp \rightarrow term\ exp' \\
&exp' \rightarrow addop\ term\ exp' \mid \varepsilon \\
&addop \rightarrow + \mid - \\
&term \rightarrow factor\ term' \\
&term' \rightarrow mulop\ factor\ term' \mid \varepsilon \\
&mulop \rightarrow * \\
&factor \rightarrow (\ exp\ ) \mid \mathbf{number}
\end{aligned}
$$

Left Recursion Removal

M(*N*, *T*)	(	**number**	)	+	-	*	$
exp	*exp* → *term exp′*	*exp* → *term exp′*					
exp′			*exp′* → ε	*exp′* → *addop term exp′*	*exp′* → *addop term exp′*		*exp′* → ε
addop				*addop* → +	*addop* → -		
term	*term* → *factor term′*	*term* → *factor term′*					
term′			*term′*→ ε	*term′* → ε	*term′* → ε	*term′* → *mulop factor term′*	*term′* → ε
mulop						*mulop* → *	
factor	*factor* → (*exp*)	*factor* → number					

Left Recursion Removal

- parse tree for the expression “**3 – 4 – 5**”



- Left associativity is not preserved.

Left Factoring

- Two grammar rules share a common prefix
  - $A \rightarrow \alpha\beta \mid \alpha\gamma$

- Left factoring
  - $A \rightarrow \alpha A'$
  - $A' \rightarrow \beta \mid \gamma$

Left Factoring

- **Examples**
  - *stmt-sequence* → *stmt* **;** *stmt-sequence* | *stmt*
  - *stmt* → **s**
  - *stmt-sequence* → *stmt stmt-seq′*
  - *stmt-seq′* → **;** *stmt-sequence* | ε
- **Left recursion removal**
  - *stmt-sequence* → *stmt-sequence* **;** *stmt* | *stmt*
  - *stmt* → **s**
  - *stmt-sequence* → *stmt stmt-seq′*
  - *stmt-seq′* → **;** *stmt stmt-seq′* | ε

Left Factoring

- **Examples**

  - *if-stmt* → **if (** *exp* **)** *statement* |
    if (*exp* **)** *statement* **else** *statement*

$$\Downarrow$$

  - *if-stmt* → **if (** *exp* **)** *statement else-part*
  - *else-part* → **else** *statement* | ε

Left Factoring

- **Examples**

  - $exp \rightarrow term + exp \mid term$

$$\Downarrow$$

  - $exp \rightarrow term\ exp'$
  - $exp' \rightarrow + exp \mid \varepsilon$

$$\Downarrow$$

  - $exp \rightarrow term\ exp'$
  - $exp' \rightarrow + term\ exp' \mid \varepsilon$

Left Factoring

- **Examples**
  - *statement* → *assign-stmt* | *call-stmt* | ***other***
  - *assign-stmt* → ***identifier*** **:=** *exp*
  - *call-stmt* → ***identifier*** **(** *exp-list* **)**

LL(1) grammar?

⇓

  - *statement* → ***identifier*** **:=** *exp* | ***identifier*** **(** *exp-list* **)** | ***other***

⇓

  - *statement* → ***identifier*** *statement′* | ***other***
  - *statement′* → **:=** *exp* | **(** *exp-list* **)**

First Sets

First(A) 의미 - A가 만나는 처음 ternminal symbol

- The first set is defined on a grammar symbol X or a string $X_1X_2\ldots X_n$.
- **First(X)** for a grammar symbol X.
  - If X is a terminal or ε, First(X) = **{X}.**
  - If X is a nonterminal, for each grammar rule $X \rightarrow X_1X_2\ldots X_n$, First($X$) includes **First($X_1X_2\ldots X_n$).**
    - *exp* → *term addop exp* | *factor*
    - First(*exp*) = First(*term addop exp*) ∪ First(*factor*)

First Sets

- **First($X_1X_2 \ldots X_n$)** for a string $X_1X_2 \ldots X_n$.
  - If there are no ε-*productions*, First($X_1X_2 \ldots X_n$) = **First(X_1).**
    - First(*exp addop term*) = First(*exp*)
    - First(**;** *stmt*) = First(**;**) ={**;**}
  - If there are some ε-*productions*,
    - First($X_1X_2 \ldots X_n$) includes **First(X_1) - {ε}.**
    - If First(X_1) includes ε, First(X) also includes **First(X_2) - {ε}.**
    - If First(X_2) includes ε, First(X) also includes **First(X_3) - {ε}.**
    - …..
    - If all First(X_k)'s include ε, First(X) also includes ε.

책 설명 :
1. if X is a terminal or ε, then First(X) = {X}
2. if X is a nonterminal, then for each production choice X->X1X2…Xn, First(X) contains First(X1) - {ε}, if also for some i<n, all the sets First(X1),.....First(Xi) contain ε, then First(X) contains First(Xi+1) - {ε}. If all the sets First(X1),...First(Xn) contain ε, then First (X) also contains ε.

First Sets

- **Example**
  - *exp* → *exp addop term* | *term*
  - *addop* → + | -
  - *term* → *term mulop factor* | *factor*
  - *mulop* → *
  - *factor* → (*exp*) | ***number***

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

First Sets

- **Example**
  - *statement* → *if-stmt* | ***other***
  - *if-stmt* → **if (** *exp* **)** *statement else-part*
  - *else-part* → **else** *statement* | ε
  - *exp* → 0 | 1

First(*statement*) = { if , other }
First(*if-stmt*) = { if }
First(*else-part*) = { else , ε }
First(*exp*) = { 0 , 1 }

First Sets

- **Example**
  - *stmt-sequence* → *stmt stmt-seq′*
  - *stmt-seq′* → **;** *stmt-sequence* | ε
  - *stmt* → **s**

First(*stmt-sequence*) = {**s**}
First(*stmt-seq′*) = {**;**, ε}
First(*stmt*) = {**s**}

Follow Sets

Folllow(A) 의미 : A를 따라오는 처음 terminal symbol (A 뒤 symbol의 firstset)

- The follow set is defined on a nonterminal ***A***.

- **Follow(*A*)** for a nonterminal *A*.
  - If *A* is a start symbol, Follow(*A*) includes **$.**
  - If there is $B \rightarrow \alpha\ A\ \gamma$, Follow(*A*) includes **First(γ)-{ε}**.
  - If there is $B \rightarrow \alpha\ A\ \gamma$ such that $\varepsilon \in$ First(γ),
    Follow(*A*) includes **Follow(*B*)**.

Follow Sets

- **Example**
  - *exp* → *exp addop term* | *term*
  - *addop* → + | -
  - *term* → *term mulop factor* | *factor*
  - *mulop* → *
  - *factor* → **(***exp***)** | ***number***

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

Follow(*exp*) = {$,), +, -}
Follow(*addop*) = {(,*number*}
Follow(*term*) = {*,$,),+,-}
Follow(*mulop*) = {(, *number*}
Follow(*factor*) = {*,$,),+,-}

Follow Sets

- **Example**
  - *statement* → *if-stmt* | ***other***
  - *if-stmt* → **if (** *exp* **)** *statement else-part*
  - *else-part* → **else** *statement* | ε
  - *exp* → 0 | 1

First(*statement*) = {***other***,**if**}
First(*if-stmt*) = {**if**}
First(*else-part*) = {**else**,ε}
First(*exp*) = {**0,1**}

Follow(*statement*) = {$, **else**}
Follow(*if-stmt*) = {$, **else**}
Follow(*else-part*) = {$, **else**}
Follow(*exp*) = {**)**}

Follow Sets

- **Example**
  - *stmt-sequence* → *stmt stmt-seq′*
  - *stmt-seq′* → **;** *stmt-sequence* | ε
  - *stmt* → **s**

First(*stmt-sequence*) = {**s**}
First(*stmt-seq′*) = {**;**,ε}
First(*stmt*) = {**s**}

Follow(*stmt-sequence*) = {$}
Follow(*stmt-seq′*) = {$}
Follow(*stmt*) = {**;**,$}

Constructing LL(1) Parsing Tables

- **LL(1) parsing table generation**
  - A table entry $M[A, \boldsymbol{a}]$ has every grammar rule $A \rightarrow \alpha$ for a nonterminal A and a terminal $\boldsymbol{a}$
    - if there is a derivation $\alpha \Rightarrow^* \boldsymbol{a}\beta$ or
    - if there is a derivation $\alpha \Rightarrow^* \boldsymbol{\varepsilon}$ and $S \Rightarrow^* \beta A \boldsymbol{a} \gamma$ for start symbol S.

Constructing LL(1) Parsing Tables

- For a grammar rule $A \rightarrow \alpha$,
  - for each token $\boldsymbol{a}$ in First(α), add it to the entry $M[A, \boldsymbol{a}]$.
- If ε is in First(α),
  - for each element $\boldsymbol{a}$ of Follow(A), add $A \rightarrow \alpha$ to the entry $M[A, \boldsymbol{a}]$.

- A grammar in BNF is LL(1) if the following conditions are satisfied.
  - For every production $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$,
    First(α_i) $\cap$ First(α_j) is empty for all $i \neq j$.
  - For every nonterminal A such that First(A) contains ε,
    First(A) $\cap$ Follow(A) is empty.

Constructing LL(1) Parsing table

statement → *if-stmt* | ***other***
if-stmt → **if (** *exp* **)** *statement else-part*
else-part → **else** *statement* | ε
exp → **0** | **1**

Ft(*st..*) = {***other***, **if**}
Ft(*if-stmt*) = {**if**}
Ft(*else..*) = {**else**,ε}
Ft(*exp*) = {**0**,**1**}

Fw(*st..*) = {$, **else**}
Fw(*if-stmt*) = {$, **else**}
Fw(*else..*) = {$, **else**}
Fw(*exp*) = {**)**}

***M*[*N*,*T*]**	**if**	***other***	**else**	**0**	**1**	**$**
statement	*statement* → *if-stmt*	*statement* → ***other***				
if-stmt	*if-stmt* → **if (** *exp* **)** *statement else-part*					
else-part			*else-part* → **else** *statement* *else-part* → ε			*else-part* → ε
exp				*exp*→0	*exp*→1	

Constructing LL(1) Parsing Tables

- **Example**
  - *stmt-sequence* → *stmt stmt-seq′*
  - *stmt-seq′* → **;** *stmt-sequence* | ε
  - *stmt* → **s**

First(*stmt-sequence*) = {**s**}
First(*stmt-seq′*) = {**;**,ε}
First(*stmt*) = {**s**}

Follow(*stmt-sequence*) = {\$}
Follow(*stmt-seq′*) = {\$}
Follow(*stmt*) = {**;**,\$}

***M*[*N*,*T*]**	**s**	**;**	**$**
stmt-sequence	*stmt-sequence* → *stmt stmt-seq′*		
stmt	*stmt* → **s**		
stmt-seq′		*stmt-seq′* → **;** *stmt-sequence*	*stmt-seq′* → ε

Constructing LL(1) Parsing table

- **Examples 4.15**

$exp \rightarrow term\ exp'$

$exp' \rightarrow addop\ term\ exp' \mid \varepsilon$

$addop \rightarrow + \mid -$

$term \rightarrow factor\ term'$

$term' \rightarrow mulop\ factor\ term' \mid \varepsilon$

$mulop \rightarrow *$

$factor \rightarrow (\ exp\ ) \mid \mathbf{number}$

$\mathrm{Ft}(exp) = \{(,\ number\}$
$\mathrm{Ft}(exp') = \{+,-,\varepsilon\}$
$\mathrm{Ft}(addop) = \{+,-\}$
$\mathrm{Ft}(term) = \{(,\ number\}$
$\mathrm{Ft}(term') = \{*,\varepsilon\}$
$\mathrm{Ft}(mulop) = \{*\}$
$\mathrm{Ft}(factor) = \{(,\ number\}$

$\mathrm{Fw}(exp) = \{\$,)\}$
$\mathrm{Fw}(exp') = \{\$,)\}$
$\mathrm{Fw}(addop) = \{(,number\}$
$\mathrm{Fw}(term) = \{\$,),+,-\}$
$\mathrm{Fw}(term') = \{\$,),+,-\}$
$\mathrm{Fw}(mulop) = \{(,number\}$
$\mathrm{Fw}(factor) = \{\$,),+,-,*\}$

Constructing LL(1) Parsing table

M(N, T)	**(**	**number**	**)**	**+**	**-**	*****	**$**
exp	*exp* → *term exp′*	*exp* → *term exp′*					
exp′			*exp′* → ε	*exp′* → *addop term exp′*	*exp′* → *addop term exp′*		*exp′* → ε
addop				*addop* → +	*addop* → -		
term	*term* → *factor term′*	*term* → *factor term′*					
term′			*term′* → ε	*term′* → ε	*term′* → ε	*term′* → *mulop factor term′*	*term′* → ε
mulop						*mulop* → *	
factor	*factor* → (*exp*)	*factor* → number					

Extending the Lookahead: LL(k) Parseres

- Lookahead k symbols
- The parsing table becomes much larger.
  - The number of columns increases exponentially with k.
- However, LL(k) parsing is not so powerful.
  - A grammar with left recursion is never LL(k) for any large k.

Error Recovery

- **Recognizer**
  - **A parser to check a program is syntactically correct or not.**
- **Error detection**
  - **Determine the location where an error has occurred as closely as possible.**
- **Error correction**
  - **Try to parse as much of the code as possible.**
  - **Avoid the error cascade problem**
  - **Avoid infinite loops on errors without consuming any input.**

Error Recovery in Recursive-Descent Parsers

- **Errors**
  - **Insertion**
  - **Deletion**
  - **Change**

- **Error recovery in recursive-descent parsers.**
  - **Panic mode**
    - **Provide each recursive procedure with an extra parameter consisting of a set of synchronizing tokens.**
    - **If an error is encountered, the parser scans ahead, throwing away tokens until one of the synchronizing set of tokens is seen.**
    - **Synchronizing tokens: Follow sets and First sets.**

Error Recovery in LL(1) Parsers

- **Error recovery in LL(1) parsers**
  - **Error occurs when the input token is not in First(A) where A is at the top of the stack.**

  - **Panic mode can be used.**
    - **Additional stack is needed to keep the synchset parameters.**
      - **because LL(1) parsing is not recursive.**

Error Recovery in LL(1) Parsers

- **Build the synchronizing tokens into the LL(1) parsing table.**
  - **Pop**
    - **Pop A from the stack**
  - **Scan**
    - **Successively pop tokens from the input until a token is seen for which we can restart the parse.**
  - **Push**
    - **Push a new nonterminal onto the stack.**

Error Recovery in LL(1) Parsers

***M*(*N*, *T*)**	**(**	***number***	**)**	**+**	**-**	*****	**$**
exp	*exp* → *term exp′*	*exp* → *term exp′*	pop	scan	scan	scan	pop
exp′	scan	scan	*exp′* → ε	*exp′* → *addop term exp′*	*exp′* → *addop term exp′*	scan	*exp′* → ε
addop	pop	pop	scan	*addop* → +	*addop* → -	scan	pop
term	*term* → *factor term′*	*term* → *factor term′*	pop	pop	pop	scan	pop
term′	scan	scan	*term′*→ ε	*term′* → ε	*term′* → ε	*term′* → *mulop factor term′*	*term′* → ε
mulop	pop	pop	scan	scan	scan	*mulop* → *	pop
factor	*factor* → (*exp*)	*factor* → ***number***	pop	pop	pop	pop	pop

Error Recovery in LL(1) Parsers

Parsing stack	Input	Action
$\$\ E'T')\ E'T$	***) $**	scan (error)
$\$\ E'T')\ E'T$	**) $**	pop (error)
$\$\ E'T')\ E'$	**) $**	$\boldsymbol{E'} \rightarrow \varepsilon$
$\$\ E'T')$	**) $**	match
$\$\ E'T'$	**$**	$\boldsymbol{T'} \rightarrow \varepsilon$
$\$\ E'$	**$**	$\boldsymbol{E'} \rightarrow \varepsilon$
$\$$	**$**	accept

Syntax Tree Construction in LL(1) Parsing

Parsing stack	Input	Action	Value stack
$\$ E$	$3+4+5\ \$$	$E \rightarrow n\, E'$	$\$$
$\$ E'\, n$	$3+4+5\ \$$	match / push	$\$$
$\$ E'$	$+4+5\ \$$	$E' \rightarrow +\, n\, \#\, E'$	$3\ \$$
$\$ E'\, \#\, n\, +$	$+4+5\ \$$	match	$3\ \$$
$\$ E'\, \#\, n$	$4+5\ \$$	match / push	$3\ \$$
$\$ E'\, \#$	$+5\ \$$	addstack	$4\ 3\ \$$
$\$ E'$	$+5\ \$$	$E' \rightarrow +\, n\, \#\, E'$	$7\ \$$
$\$ E'\, \#\, n\, +$	$+5\ \$$	match	$7\ \$$
$\$ E'\, \#\, n$	$5\ \$$	match / push	$7\ \$$
$\$ E'\, \#$	$\$$	addstack	$5\ 7\ \$$
$\$ E'$	$\$$	$E' \rightarrow \varepsilon$	$12\ \$$
$\$$	$\$$	accept	$12\ \$$