Prof. Radu Prodan

SYNTACTIC ANALYSIS
TOP-DOWN PARSING

Overview

- Top-down parsing
  - Starts with start symbol and follows leftmost derivation steps
  - Traverses parse tree in pre-order from root to leaves
- Predictive parsers
  - Choose next grammar rule using one or more lookahead tokens
- Backtracking parsers
  - Try different grammar rule possibilities
  - Back up in input if one possibility fails
  - Slow and unsuitable for practical compilers

Predictive Top-Down Parsing

- Recursive-descendant parsing
  - Ad-hoc, handwritten for each input grammar

- LL(1) parsing
  - Automatically-generated
  - Process input from **Left** to right, builds a **Leftmost** derivation and uses **1** lookahead symbol

Agenda

- **Recursive-descendant parsing**
- LL(1) parsing

Recursive Descendent Parsing

- Nonterminals are parsed by a separate procedure
  - Calls other parsing procedures in correct sequence given by body of its BNF definition

- Terminals are parsed by a `match` procedure
  - Receives expected token parameter as input
  - Checks if next input token is identical with expected token parameter and consumes it if it succeeds
  - Gives an error if not

- One global **lookahead** variable keeps next input token

Arithmetic Expression Grammar

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

```
TokenType token ;

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

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

Arithmetic Expression Grammar (2)

- *exp* → *exp addop term* | *term*
  - Calling first *exp* leads to immediate recursive loop
  - *exp* and *term* can begin with same tokens: ***number*** or **(**

- Translate grammar into EBNF

exp → *term* { *addop term* }
term → *factor* { *mulop factor* }

- Eliminate *addop* and *mulop* nonterminals that only match tokens (operators)

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

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

Arithmetic Expression Calculation

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

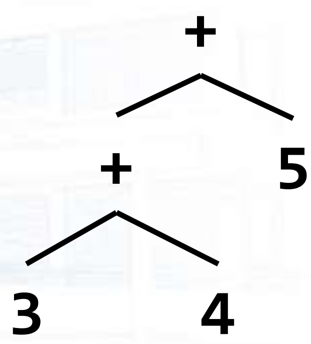
exp → *term* { *addop term* }

term → *factor* { *mulop factor* }

- Left associativity implied in EBNF definition

Syntax Tree for Arithmetic Expressions

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



exp → *term* { *addop* *term* }

term → *factor* { *mulop* *factor* }

If Statement Grammar

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

- EBNF grammar

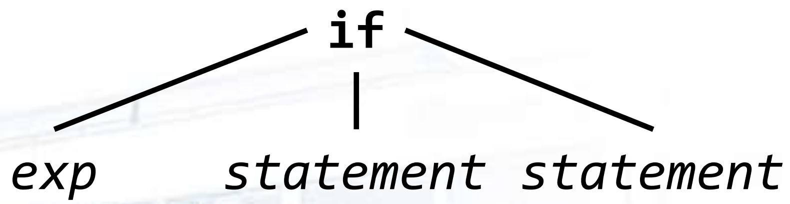
if-stmt → **if** (*exp*) *statement*
  [**else** *statement*]

- Parser uses most closely nested disambiguating rule

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

Syntax Tree for If Statement

```
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 ;
```



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

Recursive Descendent Parsing Problems

- It may be difficult to convert a BNF grammar into EBNF
  – Solution: left recursion removal
- Predictive parser that needs only one lookahead character
  – Solution: left factoring
- Recursive-descendent parsers are powerful but ad-hoc and handwritten
  – Solution: automatic LL parser generator

Agenda

- Recursive-descendant parsing
  - **Left recursion removal**
  - Left factoring
- LL(1) parsing

Left Recursion Removal

- Immediate left recursion

- $A \rightarrow A\ \alpha \mid \beta$
  - α and β are strings of terminals and nonterminals
  - β does not begin with A
  - $L(G) = \{ \beta\ \alpha^n \mid n \geq 0 \}$

- Equivalent grammar that uses right recursion

$$A \rightarrow \beta\ A'$$

$$A' \rightarrow \alpha\ A' \mid \varepsilon$$

Immediate Left Recursion Removal

- Left recursive grammar

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

  - $\beta_1, \beta_2, \ldots, \beta_m$ do not begin with an A

- Removed left recursion

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

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

Indirect Left Recursion Removal

$$A \rightarrow B\ a \mid c$$
$$B \rightarrow A\ b \mid d$$

- Transform all indirect left recursions into immediate left recursions
- Choose an arbitrary order of nonterminals $A_1, \ldots, A_m$
- Eliminate all rules of form $A_i \rightarrow A_j\ \gamma$, with $j \leq i$
  - Replace A_j by its definition

Indirect Left Recursive	Direct Left Recursive	Right Recursive
$A_1 \rightarrow A_2\ a \mid c$ $A_2 \rightarrow A_1\ b \mid d$	$A_1 \rightarrow A_2\ a \mid c$ $A_2 \rightarrow A_2\ a\ b \mid c\ b \mid d$	$A_1 \rightarrow A_2\ a \mid c$ $A_2 \rightarrow c\ b\ A_2' \mid d\ A_2'$ $A_2' \rightarrow a\ b\ A_2' \mid \varepsilon$

Indirect Left Recursion Removal Algorithm

```
(* for all nonterminals in a well defined ranking *)
for i:= 1 to m do
(* for all nonterminal with a smaller rank *)
  for j:= 1 to i-1 do
     Replace each grammar rule A_i → A_j β by rule
          A_i → α_1 β | α_2 β | . . . | α_k β,
          where A_j → α_1 | α_2 | . . . | α_k
  Eliminate direct left recursions of A_i
```

- No cycles and ε-productions

Indirect Left Recursion Removal Example

$$A_1 \rightarrow A_2\ a \mid A_1\ a \mid c$$
$$A_2 \rightarrow A_2\ b \mid A_1\ b \mid d$$

- Left recursion does not change language, but changes grammar
- Changes parse trees and complicates parser

Outer loop	Inner loop	Action	Grammar
$i = 1$	*Inner loop does not execute*	*Remove immediate left recursion on* A_1	$A_1 \rightarrow A_2\ a\ A_1' \mid c\ A_1'$ $A_1' \rightarrow a\ A_1' \mid \varepsilon$ $A_2 \rightarrow A_2\ b \mid A_1\ b \mid d$
$i = 2$	$j = 1$	*Eliminate rule* $A_2 \rightarrow A_1\ b$	$A_1 \rightarrow A_2\ a\ A_1' \mid c\ A_1'$ $A_1' \rightarrow a\ A_1' \mid \varepsilon$ $A_2 \rightarrow A_2\ b \mid A_2\ a\ A_1'\ b \mid c\ A_1'\ b \mid d$
$i = 2$	*Inner loop done*	*Remove left recursion on* A_2	$A_1 \rightarrow A_2\ a\ A_1' \mid c\ A_1'$ $A_1' \rightarrow a\ A_1' \mid \varepsilon$ $A_2 \rightarrow c\ A_1'\ b\ A_2' \mid d\ A_2'$ $A_2' \rightarrow b\ A_2' \mid a\ A_1'\ b\ A_2' \mid \varepsilon$

Arithmetic Expression Grammar

- Left recursive grammar

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

- Right recursive grammar

  $exp \rightarrow term\ exp'$

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

Right Recursive Expression Parser

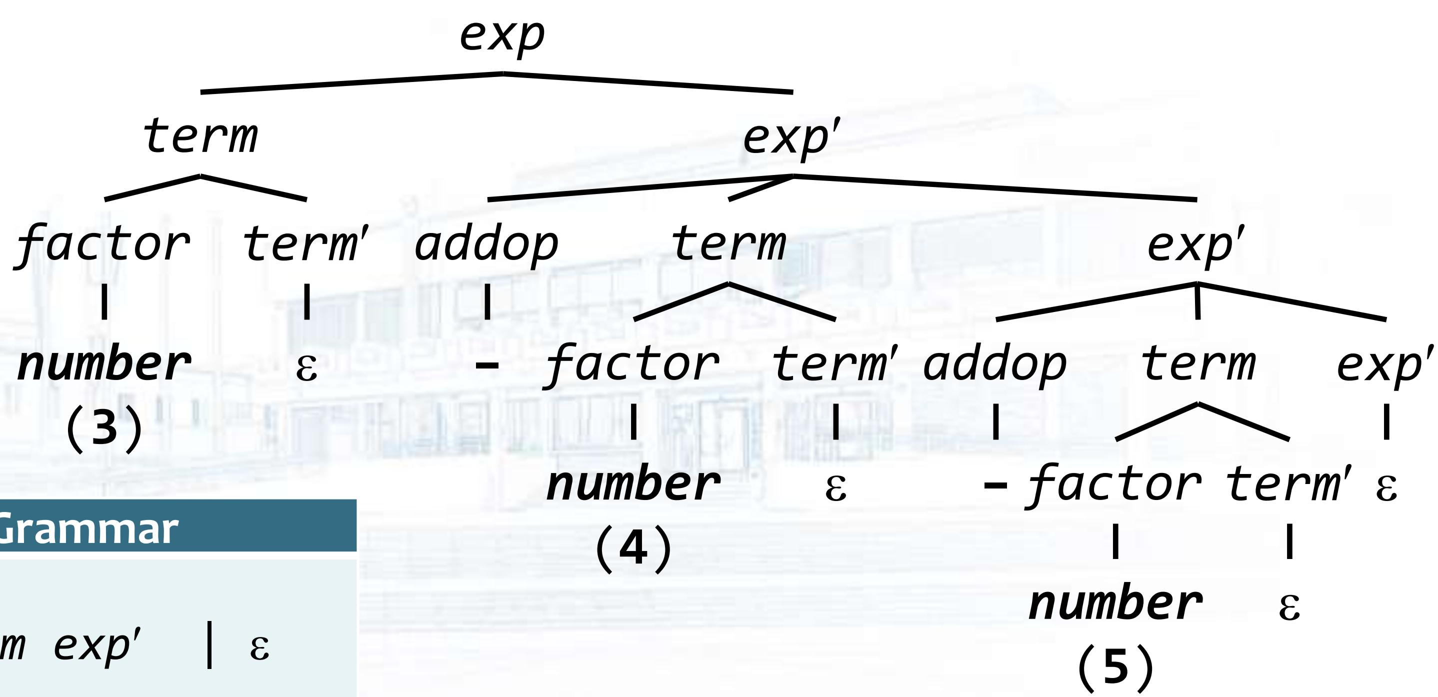
Left Recursive Grammar	Equivalent Right Recursive Grammar
exp → *exp addop term* | *term* *addop* → **+** | **-** *term* → *term multop factor* | *factor* *mulop* → ***** *factor* → **(** *exp* **)** | ***number***	*exp* → *term exp′* *exp′* → *addop term exp′* | ε *addop* → **+** | **-** *term* → *factor term′* *term′* → *mulop factor term′* | ε *mulop* → ***** *factor* → **(** *exp* **)** | ***number***

```
procedure exp ;
begin
   term () ;
   exp′ () ;
end exp ;
```

```
procedure exp′ ;
begin
   case token of
   + :  match (+) ;
        term () ;
        exp′ () ;
   - :  match (-) ;
        term () ;
        exp′ () ;
   end case ;
end exp′ ;
```

Loss of Left Associativity

- Parse tree for **3 – 4 – 5**



Expression Grammar

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

Left Recursive Parser

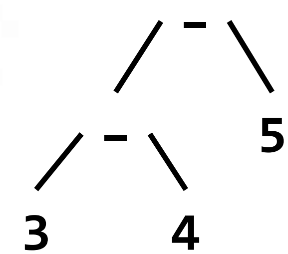
```
function exp : integer ;
var  temp : integer ;
begin
   temp := term () ;
   return exp′ (temp) ;
end exp ;

function exp′ (valsofar : integer) : integer ;
begin
   if token = + or token = - then
        match (token) ;
        case token of
        + :     valsofar := valsofar + term () ;
        - :     valsofar := valsofar - term () ;
        end case ;
        return exp′ (valsofar) ;
   else return valsofar ;
end exp′ ;
```

Expression Grammar

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

- **3 – 4 – 5**



Agenda

- Recursive-descendant parsing
  - Left recursion removal
  - **Left factoring**
- LL(1) parsing

Left Factoring

- Two or more grammar rule choices share a common prefix string
  - $A \rightarrow \alpha\ \beta \mid \alpha\ \gamma$
- More than one lookahead character necessary
- Rewrite rule as two rules with α as common factor
  - $A \rightarrow \alpha\ A'$
  - $A' \rightarrow \beta \mid \gamma$
- **Longest common substring** α in different non-terminal definitions

Arithmetic Expression Grammar

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

- After left factoring

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

- Replacing *exp* with *term exp′* in second rule gives identical results as after left recursion removal

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

Grammar of If Statements

$$\textit{if-stmt} \rightarrow \textbf{if} \ (\ \textit{exp} \ ) \ \textit{statement}$$
$$| \ \textbf{if} \ (\ \textit{exp} \ ) \ \textit{statement} \ \textbf{else} \ \textit{statement}$$

- After left factoring

$$\textit{if-stmt} \rightarrow \textbf{if} \ (\ \textit{exp} \ ) \ \textit{statement} \ \textit{else-part}$$
$$\textit{else-part} \rightarrow \textbf{else} \ \textit{statement} \ | \ \varepsilon$$

Left Factoring Algorithm

while *there are changes to grammar* **do**
  for $\forall\ A \in N \wedge A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \in P$ **do**
    let α *be a prefix of maximum length that is shared by two or more production choices for* A
    if $\alpha \neq \varepsilon$ **then**
      suppose that $\alpha_1, \ldots, \alpha_k$ *share* α*, so that*
        $A \rightarrow \alpha\ \beta_1 \mid \ldots \mid \alpha\ \beta_k \mid \alpha_{k+1} \mid \ldots \mid \alpha_n$,
        β_j*'s share no common prefix* ($j \in [1..k]$)
        and $\alpha_{k+1}, \ldots, \alpha_n$ *do not share* α
      replace rule $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$
        by rules:
        $A \rightarrow \alpha\ A' \mid \alpha_{k+1} \mid \ldots \mid \alpha_n$
        $A' \rightarrow \beta_1 \mid \ldots \mid \beta_k$

Grammar of Statement Sequences

- Right recursive form

  stmt-sequence → *stmt* **;** *stmt-sequence* | *stmt*

  stmt → **s**

- After left factoring

  stmt-sequence → *stmt* *stmt-seq′*

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

- Left recursive form

  stmt-sequence → *stmt-sequence* **;** *stmt* | *stmt*

  stmt → **s**

- Left recursion removal

  stmt-sequence → *stmt* *stmt-seq′*

  stmt-seq′ → **;** *stmt* *stmt-seq′* | ε

Agenda

- Recursive-descendant parsing
  - Left recursion removal
  - Left factoring

- **LL(1) parsing**

LL(1) Parsing Overview

- Requires a **right recursive** and **left factored** grammar
- Uses an explicit stack instead of recursive calls
- Mark bottom of stack with dollar ($) character
- **Match** a token on top of the stack with next input token
- **Generate** replaces a nonterminal A at top of stack by string α using grammar rule $A \rightarrow \alpha$
  - α pushed onto stack in reversed order of symbols

	Parsing stack	Input	Action
1	`$ Start symbol`	`Input string`	
	. . .	. . .	
	. . .	. . .	
	$	$	***accept***

Balanced Parentheses Grammar

$$S \rightarrow (\; S \;) \; S$$
$$\quad | \; \varepsilon$$

	Parsing Stack	Input	Action
1	$\$ \; S$	$(\;) \; \$$	$S \rightarrow (\; S \;) \; S$
2	$\$ \; S \;) \; S \; ($	$(\;) \; \$$	*match*
3	$\$ \; S \;) \; S$	$) \; \$$	$S \rightarrow \varepsilon$
4	$\$ \; S \;)$	$) \; \$$	*match*
5	$\$ \; S$	$\$$	$S \rightarrow \varepsilon$
6	$\$$	$\$$	***accept***

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

If-Statement Grammar

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			*else-part* → ε
exp				*exp* → **0**	*exp* → **1**	

LL(1) Parsing Actions for Grammar of if-Statements

Parsing Stack	Input	Action
$ *statement*	`if(0) if(1)` ***other*** `else` ***other*** $	*statement* → *if-stmt*
$ *if-stmt*	`if(0) if(1)` ***other*** `else` ***other*** $	*if-stmt* → **if** (*exp*) *statement else-part*
$ *else-part statement*) *exp* (**if**	`if(0) if(1)` ***other*** `else` ***other*** $	*match*
$ *else-part statement*) *exp* (	`(0) if(1)` ***other*** `else` ***other*** $	*match*
$ *else-part statement*) *exp*	`0) if(1)` ***other*** `else` ***other*** $	*exp* → **0**
$ *else-part statement*) **0**	`0) if(1)` ***other*** `else` ***other*** $	*match*
$ *else-part statement*)	`) if(1)` ***other*** `else` ***other*** $	*match*
$ *else-part statement*	`if(1)` ***other*** `else` ***other*** $	*statement* → *if-stmt*
$ *else-part if-stmt*	`if(1)` ***other*** `else` ***other*** $	*if-stmt* → **if** (*exp*) *statement else-part*
$ *else-part else-part statement*) *exp* (**if**	`if(1)` ***other*** `else` ***other*** $	*match*
$ *else-part else-part statement*) *exp* (	`(1)` ***other*** `else` ***other*** $	*match*
$ *else-part else-part statement*) *exp*	`1)` ***other*** `else` ***other*** $	*exp* → **1**
$ *else-part else-part statement*) **1**	`1)` ***other*** `else` ***other*** $	*match*
$ *else-part else-part statement*)	`)` ***other*** `else` ***other*** $	*match*
$ *else-part else-part statement*	***other*** `else` ***other*** $	*statement* → ***other***
$ *else-part else-part* ***other***	***other*** `else` ***other*** $	*match*
$ *else-part else-part*	`else` ***other*** $	*else-part* → **else** *statement*
$ *else-part statement* **else**	`else` ***other*** $	*match*
$ *else-part statement*	***other*** $	*statement* → ***other***
$ *else-part* ***other***	***other*** $	*match*
$ *else-part*	$	*else-part* → ε
$	$	***accept***

LL(1) Parsing Algorithm

```
(* assumes $ marks bottom of stack and end of input *)
while top(parsing stack) ≠ $ ∧ token ≠ $ do
   if top(parsing stack) = a ∈ T ∧ token = a
   then (* match *)
        pop(a, parsing stack) ;
        token = getToken() ;
   else if top(parsing stack) = A ∈ N ∧ token = a ∈ T ∧
                                  ∧ A → X1X2 … Xn ∈ M[A, a]
        then (* generate *)
               pop(A, parsing stack) ;
               for i := n downto 1 do
                      push(Xi, parsing stack) ;
        else error ;
if top(parsing stack) = $ ∧ token = $
then accept
else error ;
```

???

LL(1) Parsing Table

- Context-free grammar: $G = (T, N, P, S)$

- **Parsing table** indexed by nonterminals and terminals which contains production rules to use when
  - Nonterminal is on top of stack
  - Terminal is next in input

- A production $(A \to \alpha) \in M[A, a]$ in two cases:
  - $(\exists\, \alpha \Rightarrow^* a\beta) \wedge a \in T$
    - α starts with terminal a: **$a \in$ First(α)**
  - $(\exists\, \alpha \Rightarrow^* \varepsilon) \wedge (S\$ \Rightarrow^* \beta A a \gamma) \wedge a \in T \cup \$$
    - A is followed by terminal a if it can disappear: **$a \in$ Follow(A)**

$S \to (\; S \;) \; S \mid \varepsilon$

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

LL(1) Parsing Table Construction Algorithm

```
for ∀ A ∈ N ∧ ∀ A → α ∈ P do
   for ∀ a ∈ First(α) do
      M[A, a] = M[A, a] ∪ { A → α }
   if ε ∈ First(α) then
      for ∀ a ∈ Follow(A) do
         M[A, a] = M[A, a] ∪ { A → α }
```

- If $(A \to \alpha \in P) \wedge (\exists\, \alpha \Rightarrow^* a\beta) \wedge (a \in T) \Rightarrow M[A, a] = M[A, a] \cup \{ A \to \alpha \}$
  - $a \in \text{First}(\alpha)$

- If $(A \to \alpha \in P) \wedge (\exists\, \alpha \Rightarrow^* \varepsilon) \wedge (S\,\$ \Rightarrow^* \beta\, A\, a\, \gamma) \wedge (a \in T \cup \$) \Rightarrow$
  $\Rightarrow M[A, a] = M[A, a] \cup \{ A \to \alpha \}$
  - $a \in \text{Follow}(A)$

Agenda

- Recursive-descendant parsing
  – Left recursion removal
  – Left factoring

- LL(1) parsing
  – **FIRST sets**
  – FOLLOW sets
  – Parsing table
  – LL(1) grammars

- Error recovery

First Sets

- $G = (T, N, P, S)$

- $X \in T \cup N \cup \varepsilon$

- Set **First(X)** $\subset T \cup \varepsilon$ is defined as follows:
  - If $X \in T \cup \varepsilon \Rightarrow \text{First}(X) = \{ X \}$
  - If $X \in N$, then $\forall\, X \rightarrow X_1 X_2 \ldots X_n \in P \Rightarrow$
    - $\text{First}(X_1) - \varepsilon \subset \text{First}(X)$
    - If $\varepsilon \in \text{First}(X_1) \wedge \ldots \wedge \varepsilon \in \text{First}(X_i) \wedge i < n \Rightarrow \text{First}(X_{i+1}) - \{ \varepsilon \} \subset \text{First}(X)$
    - If $\varepsilon \in \text{First}(X_1) \wedge \ldots \wedge \varepsilon \in \text{First}(X_n) \Rightarrow \varepsilon \in \text{First}(X)$

Integer Expression Grammar: First Sets Computation

$exp \rightarrow exp\ addop\ term \mid term$

$term \rightarrow term\ mulop\ factor \mid factor$

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

$addop \rightarrow + \mid -$

$mulop \rightarrow *$

Grammar Rule	Iteration 1	Iteration 2	Iteration 3
$exp \rightarrow exp\ addop\ term$			
$exp \rightarrow term$			First(exp) = { (, **number** }
$addop \rightarrow +$	First($addop$) = { + }		
$addop \rightarrow -$	First($addop$) = { +, - }		
$term \rightarrow term\ mulop\ factor$			
$term \rightarrow factor$		First($term$) = { (, **number** }	
$mulop \rightarrow *$	First($mulop$) = { * }		
$factor \rightarrow (\ exp\ )$	First($factor$) = { (}		
$factor \rightarrow \textbf{number}$	First($factor$) = { (, **number** }		

Statement Sequence Grammar: First Sets Computation

- Left recursive

stmt-sequence → *stmt-sequence* **;** *stmt* | *stmt*
stmt → **s**

- Left factored right recursive

stmt-sequence → *stmt* *stmt-seq'*
stmt-seq' → **;** *stmt-sequence* | ε
stmt → **s**

Grammar Productions	Iteration 1	Iteration 2
stmt-sequence → *stmt* *stmt-seq'*		First(*stmt-sequence*) = = { **s** }
stmt-seq' → **;** *stmt-sequence*	First(*stmt-seq'*) = { **;** }	
stmt-seq' → ε	First(*stmt-seq'*) = { **;**, ε}	
stmt → **s**	First(*stmt*) = { **s** }	

If-Statement Grammar: First Sets Computation

statement → *if-stmt* | ***other***

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

else-part → **else** *statement* | ε

exp → **0** | **1**

Grammar Rule	Iteration 1	Iteration 2
statement → *if-stmt*		First(*statement*) = = { **if**, ***other*** }
statement → ***other***	First(*statement*) = { ***other*** }	
if-stmt → **if** **(** *exp* **)** *statement else-part*	First(*if-stmt*) = { **if** }	
else-part → **else** *statement*	First(*else-part*) = { **else** }	
else-part → ε	First(*else-part*) = { **else**, ε }	
exp → **0**	First(*exp*) = { **0** }	
exp → **1**	First(*exp*) = { **0**, **1** }	

First Set Computation Algorithm

```
for ∀ A ∈ N do
    First(A) := Φ ;

while there are changes to any First(A) do
  for ∀ A → X1 X2 … Xn do
      k := 1 ;
      continue := true ;
      while continue = true ∧ k ≤ n do
            First(A) := First(A) ∪ First(Xk) – { ε } ;
            if ε ∉ First(Xk) then
                  continue := false ;
            k := k + 1 ;
      if continue = true then
            First(A) := First(A) ∪ { ε } ;
```

Agenda

- Recursive-descendant parsing
  - Left recursion removal
  - Left factoring
- LL(1) parsing
  - FIRST sets
  - **FOLLOW sets**
  - Parsing table
  - LL(1) grammars
- Error recovery

Follow Sets

- $G = (T, N, P, S)$ and $A \in N$
- Set **Follow(A)** $\subset T \cup \$$ is defined as follows:
  - If $A = S \Rightarrow \$ \in \text{Follow}(A)$
  - If $(\exists\, B \rightarrow \alpha\, A\, \gamma \in P) \Rightarrow \text{First}(\gamma) - \varepsilon \in \text{Follow}(A)$
  - If $(\exists\, B \rightarrow \alpha\, A\, \gamma \in P) \wedge \varepsilon \in \text{First}(\gamma)$

    $\Rightarrow \text{Follow}(B) \subset \text{Follow}(A)$
    - $B \rightarrow \alpha\, A$ is a common special case
- ε is never an element of Follow set

Simple Expression Grammar: Follow Sets Computation

Grammar Rule	Iteration 1	Iteration 2
exp → *exp* *addop* *term*	Follow(*exp*) = \$ ∪ First(*addop*) = { \$, **+**, **−** } Follow(*addop*) = First(*term*) = { **(**, ***number*** } Follow(*term*) = Follow(*exp*) = { \$, **+**, **−** }	Follow(*term*) ∪= Follow(*exp*) = = { \$, **+**, **−**, **)** }
exp → *term*	Follow(*term*) = Follow(*exp*) = { \$, **+**, **−** }	Follow(*term*) ∪= Follow(*exp*) = = { \$, **+**, **−**, **)** }
term → *term* *mulop* *factor*	Follow(*term*) = First(*mulop*) = { \$, **+**, **−**, ***** } Follow(*mulop*) = First(*factor*) = { **(**, ***number*** } Follow(*factor*) = Follow(*term*) = { \$, **+**, **−**, ***** }	Follow(*factor*) ∪= Follow(*term*) = = { \$, **+**, **−**, *****, **)** }
term → *factor*	Follow(*factor*) = Follow(*term*) = { \$, **+**, **−**, ***** }	Follow(*factor*) ∪= Follow(*term*) = = { \$, **+**, **−**, *****, **)** }
factor → **(** *exp* **)**	Follow(*exp*) = First(**)**) = { \$, **+**, **−**, **)** }	

Statement Sequence Grammar: Follow Sets Computation

Grammar Rule	Iteration 1
stmt-sequence → *stmt* *stmt-seq′*	Follow(*stmt-sequence*) = { \$ } Follow(*stmt*) = First(*stmt-seq′*) – { ε } = { **;** } Follow(*stmt*) = Follow(*stmt-sequence*) = { **;**, \$ } Follow(*stmt-seq′*) = Follow(*stmt-sequence*) = { \$ }
stmt-seq′ → **;** *stmt-sequence*	Follow(*stmt-sequence*) = Follow(*stmt-seq′*) = { \$ }

If-Statement Grammar: Follow Sets Computation

Grammar Rule	Iteration 1	Iteration 2
statement → *if-stmt*	Follow(*statement*) = { \$ } Follow(*if-stmt*) = Follow(*statement*) = { \$ }	Follow(*if-stmt*) = Follow(*statement*) = { \$, **else** }
if-stmt → **if (*exp*)** *statement* *else-part*	Follow(*exp*) = First(**)**) = { **)** } Follow(*statement*) = First(*else-part*) – { ε } = = { \$, **else** } Follow(*statement*) = Follow(*if-stmt*) = = { \$, **else** } Follow(*else-part*) = Follow(*if-stmt*) = { \$ }	Follow(*statement*) = Follow(*if-stmt*) = { \$, **else** } Follow(*else-part*) = Follow(*if-stmt*) = { \$, **else** }
else-part → **else** *statement*	Follow(*statement*) = Follow(*else-part*) = = { \$, **else** }	Follow(*statement*) = Follow(*else-part*) = { \$, **else** }
exp → **0** | **1**		

Follow Set Computation Algorithm

```
Follow(S) := $ ;
for ∀ A ∈ N – { S } do
  Follow(A) := Φ ;

while there are changes to any Follow sets do
  for ∀ A → X1 X2 … Xn ∈ P do
    for ∀ i∈[1..n] do
      Follow(Xi) := Follow(Xi) ∪
                          First(Xi+1 Xi+2 … Xn) – { ε } ;
      (* Note: if i=n, then Xi+1 Xi+2 … Xn = ε *)
      if ε ∈ First(Xi+1 Xi+2 … Xn) then
        Follow(Xi) := Follow(Xi) ∪ Follow(A) ;
```

Agenda

- Recursive-descendant parsing
  - Left recursion removal
  - Left factoring
- LL(1) parsing
  - FIRST sets
  - FOLLOW sets
  - **Parsing table**
  - LL(1) grammars
- Error recovery

Simple Expression Grammar: Parsing Table

Grammar Rule	First Set	Follow Set
exp → *exp addop term* | *term*	First(*exp*) = { (, ***number*** }	Follow(*exp*) = { $, +, –,) }
addop → **+** | **-**	First(*addop*) = { **+**, **–** }	Follow(*addop*) = { **(**, ***number*** }
term → *term mulop factor* | *factor*	First(*term*) = { (, ***number*** }	Follow(*term*) = { $, +, –,) }
mulop → *****	First(*mulop*) = { ***** }	Follow(*mulop*) = { **(**, ***number*** }
factor → **(** *exp* **)** | ***number***	First(*factor*) = { (, ***number*** }	Follow(*factor*) = { $, +, –, *,) }

M[N,T]	(	*number*	)	+	-	*	$
exp	*exp* → *exp addop term* | *term*	*exp* → *exp addop term* | *term*					
addop				*addop* → **+**	*addop* → **-**		
term	*term* → *term mulop factor* | *factor*	*term* → *term mulop factor* | *factor*					
mulop						*mulop* → *****	
factor	*factor* → **(** *exp* **)**	*factor* → ***number***					

Statement Sequence Grammar: Parsing Table

Grammar Rule	First Set	Follow Set
stmt-sequence → *stmt stmt-seq'*	First(*stmt-sequence*) = { **s** }	Follow(*stmt-sequence*) = { $ }
stmt-seq' → **;** *stmt-sequence* | ε	First(*stmt-seq'*) = { **;**, ε }	Follow(*stmt-seq'*) = { $ }
stmt → **s**	First(*stmt*) = { **s** }	Follow(*stmt*) = { **;**, $ }

M[*N*, *T*]	**s**	**;**	**$**
stmt-sequence	*stmt-sequence* → *stmt stmt-seq'*		
stmt-seq'		*stmt-seq'* → **;** *stmt-sequence*	*stmt-seq'* → ε
stmt	*stmt* → **s**		

If-Statement Grammar: Parsing Table

Grammar Rule	First Set	Follow Sets
statement → *if-stmt* | ***other***	First(*statement*) = { **if**, ***other*** }	Follow(*statement*) = { $, **else** }
if-stmt → **if** (*exp*) *statement* *else-part*	First(*if-stmt*) = { **if** }	Follow(*if-stmt*) = { $, **else** }
else-part → **else** *statement* | ε	First(*else-part*) = { **else**, ε }	Follow(*else-part*) = { $, **else** }
exp → **0** | **1**	First(*exp*) = { **0**, **1** }	Follow(*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**	

Expression Grammar: First Sets Computation

Grammar Rule	Iteration 1	Iteration 2	Iteration 3
$exp \rightarrow term\ exp'$			First(exp) = First($term$) = { **(**, **number** }
$exp' \rightarrow addop\ term\ exp'$		First(exp') = First($addop$) = { **+**, **−**, ε }	
$exp' \rightarrow \varepsilon$	First(exp') = { ε }		
$addop \rightarrow$ **+**	First($addop$) = { **+** }		
$addop \rightarrow$ **-**	First($addop$) = { **+**, **-** }		
$term \rightarrow factor\ term'$		First($term$) = First($factor$) = { **(**, **number** }	
$term' \rightarrow multop\ factor\ term'$		First($term'$) = First($mulop$) = { *****, ε }	
$term' \rightarrow \varepsilon$	First($term'$) = { ε }		
$mulop \rightarrow$ *****	First($mulop$) = { ***** }		
$factor \rightarrow$ **(** exp **)**	First($factor$) = { **(** }		
$factor \rightarrow$ **number**	First($factor$) = { **(**, **number** }		

Expression Grammar: Follow Sets Computation

Grammar Rule	Iteration 1	Iteration 2
exp → *term exp′*	Follow(*exp*) = { $ } Follow(*term*) = First(*exp′*) = { **+**, **−** } Follow(*exp′*) = Follow(*exp*) = { $ }	Follow(*exp′*) = Follow(*exp*) = { $, **)** }
exp′ → *addop term exp′*	Follow(*addop*) = First(*term*) = { **(**, ***number*** } Follow(*term*) = First(*exp′*) = { **+**, **−** } Follow(*term*) = Follow(*exp′*) = { $, **+**, **−** }	Follow(*term*) = Follow(*exp′*) = { $, **)**, **+**, **−** }
term → *factor term′*	Follow(*factor*) = First(*term′*) = { ***** } Follow(*factor*) = Follow(term) = { $, **+**, **−**, ***** } Follow(*term′*) = Follow(*term*) = { $, **+**, **−** }	Follow(term′) = Follow(term) = { $, **)**, **+**, **−** }
term′ → *mulop factor term′*	Follow(*mulop*) = First(*factor*) = { **(**, ***number*** } Follow(*factor*) = First(*term′*) = {$, **+**, **−**, ***** } Follow(*factor*) = Follow(*term′*) = { $, **+**, **−**, ***** }	Follow(*factor*) = Follow(*term′*) = { $, **)**, **+**, **−**, ***** }
factor → **(** *exp* **)**	Follow(*exp*) = First(**)**) = { $, **)** }	

Expression Grammar: LL(1) Parsing Table

Grammar Productions	First Sets	Follow Sets
$exp \to term\ exp'$	First(exp) = { (, **number** }	Follow(exp) = { \$,) }
$exp' \to addop\ term\ exp'$ | ε	First(exp') = { +, −, ε }	Follow(exp') = { \$,) }
$addop \to$ + | −	First($addop$) = { +, − }	Follow($addop$) = { (, **number** }
$term \to factor\ term'$	First($term$) = { (, **number** }	Follow($term$) = { \$,), +, − }
$term' \to mulop\ factor\ term'$ | ε	First($term'$) = { *, ε }	Follow($term'$) = { \$,), +, − }
$mulop \to$ *	First($mulop$) = { * }	Follow($mulop$) = { (, **number** }
$factor \to$ (exp) | **number**	First($factor$) = { (, **number** }	Follow($factor$) = { \$,), +, −, * }

M[N,T]	(	**number**	)	+	−	*	\$
exp	$exp \to term\ exp'$	$exp \to term\ exp'$					
exp'			$exp' \to \varepsilon$	$exp' \to addop\ term\ exp'$	$exp' \to addop\ term\ exp'$		$exp' \to \varepsilon$
$addop$				$addop \to$ +	$addop \to$ −		
$term$	$term \to factor\ term'$	$term \to factor\ term'$					
$term'$			$term' \to \varepsilon$	$term' \to \varepsilon$	$term' \to \varepsilon$	$term' \to mulop\ factor\ term'$	$term' \to \varepsilon$
$mulop$						$mulop \to$ *	
$factor$	$factor \to$ (exp)	$factor \to$ **number**					

Agenda

- Recursive-descendant parsing
  - Left recursion removal
  - Left factoring
- LL(1) parsing
  - FIRST sets
  - FOLLOW sets
  - Parsing table
  - **LL(1) grammars**
- Error recovery

LL(1) Grammar

- Grammar is LL(1) if associated LL(1) parsing table has at most one production in each table entry
  - An LL(1) grammar cannot be ambiguous

- $G = (T, N, P, S)$ is **LL(1)** if following conditions are satisfied
  - $\text{First}(\alpha_i) \cap \text{First}(\alpha_j) = \emptyset, \forall A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \wedge i, j \in [1..n] \wedge i \neq j$
  - $\text{First}(A) \cap \text{Follow}(A) = \emptyset, \forall A \in N \wedge \varepsilon \in \text{First}(A)$

Non-LL(1) Programming Language

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

- Replace *assign-stmt* and *call-stmt* by right-hand sides of their defining productions

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

- Left factoring

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

Expression Evaluation in LL(1) Parsing

- Expression grammar
  $E \rightarrow E + \boldsymbol{n} \mid \boldsymbol{n}$
- Left recursion removal
  $E \rightarrow \boldsymbol{n}\ E'$
  $E' \rightarrow + \boldsymbol{n}\ E' \mid \varepsilon$
- **Value stack**
  - Push number after each *match*
  - Add operation indicated on parsing stack by a special **pound symbol** (#)
  - Left associativity

$E \rightarrow \boldsymbol{n}\ E'$
$E' \rightarrow + \boldsymbol{n}\ \#\ E' \mid \varepsilon$

Parsing Stack	Input	Action	Value Stack
$\$\ E$	3 + 4 + 5 $	$E \rightarrow \boldsymbol{n}\ E'$	$
$\$\ E'\ \boldsymbol{n}$	3 + 4 + 5 $	*match/push*	$
$\$\ E'$	+ 4 + 5 $	$E' \rightarrow + \boldsymbol{n}\ \#\ E'$	3 $
$\$\ E'\ \#\ \boldsymbol{n}\ +$	+ 4 + 5 $	*match*	3 $
$\$\ E'\ \#\ \boldsymbol{n}$	4 + 5 $	*match/push*	3 $
$\$\ E'\ \#$	+ 5 $	*add stack*	4 3 $
$\$\ E'$	+ 5 $	$E' \rightarrow + \boldsymbol{n}\ \#\ E'$	7 $
$\$\ E'\ \#\ \boldsymbol{n}\ +$	+ 5 $	*match*	7 $
$\$\ E'\ \#\ \boldsymbol{n}$	5 $	*match/push*	7 $
$\$\ E'\ \#$	$	*add stack*	5 7 $
$\$\ E'$	$	$E' \rightarrow \varepsilon$	12 $
$	$	***accept***	12 $

Agenda

- Recursive-descendant parsing
  - Left recursion removal
  - Left factoring
- LL(1) parsing
  - FIRST sets
  - FOLLOW sets
  - Algorithm
- **Error recovery**

Error Recovery

- Recogniser
  - Determines if a program is syntactically correct
  - Displays a helpful error message
- Goals
  - Find error as soon as possible
  - Find a good place to resume parsing
  - Find as many real errors as possible
  - Avoid error cascade
  - Avoid infinite loops on errors
- Error correction or error repair
  - Find a correct program closest to wrong one

Panic Mode Error Recovery

- Recursive descendant parsers

- Synchronising tokens for each recursive procedure
  - Scan ahead (ignore input tokens) on errors until reaching one synchronising token
  - First sets and follow sets as synchronising tokens
  - First sets allow parser detect errors early in parse

- In a recursive procedure parsing nonterminal *N*
  - Check if next token is in First(*N*)
  - If an error happens, ignore tokens until First(*N*) $\cup$ Follow(*N*)

Recursive Descendant Error Recovery in Simple Expression Grammar

```
procedure checkinput ( firstset, followset ) ;
begin
    if ( token ∉ firstset ) then
        error ;
        scanto (firstset ∪ followset ) ;
    end if ;
end checkinput ;
```

```
procedure scanto ( syncset ) ;
begin
    while token ∉ syncset ∪ { $ } do
        token = getToken () ;
    end while ;
end scanto ;
```

```
procedure exp ( syncset ) ;
begin
    checkinput ( { (, number }, syncset ) ;
    if ( token ∈ { (, number } ) then
        term ( syncset ∪ { +, – } ) ;
        while token = + or token = – do
            match ( token ) ;
            term ( syncset ∪ { +, – } ) ;
        end while ;
        checkinput ( syncset, { (, number } ) ;
    end if ;
end exp ;
```

```
procedure factor ( syncset ) ;
begin
    checkinput ( { (, number }, syncset ) ;
    if ( token ∈ { (, number } ) then
        case token of
        ( :         match ( ( ) ;
                    exp ( { ) } ) ;
                    match ( ) ) ;
        number :    match ( number ) ;
        else error ;
        end case ;
        checkinput ( syncset, { (, number } ) ;
    end if ;
end factor ;
```

Error Recovery in LL(1) Parsers

- Nonterminal A on top of stack
- Input token T

- If $M[A, T] = \emptyset \Rightarrow$ **Error**
  - $T \notin \text{First}(A)$
  - $T \notin \text{Follow}(A)$, if $\varepsilon \in \text{First}(A)$

- $T = \$ \vee T \in \text{Follow}(A)$
  - **Pop** A from stack

- $T \neq \$ \wedge T \notin \text{First}(A) \cup \text{Follow}(A)$
  - **Scan** input until $T \in \text{First}(A) \cup \text{Follow}(A)$

- **Push** a new nonterminal onto stack

Parsing Table with Error Recovery in Expression Grammar

Grammar Productions	First Sets	Follow Sets
exp → *term exp′*	First(*exp*) = { (, **number** }	Follow(*exp*) = { $,) }
exp′ → *addop term exp′* | ε	First(*exp′*) = { +, −, ε }	Follow(*exp′*) = { $,) }
addop → + | −	First(*addop*) = { +, − }	Follow(*addop*) = { (, **number** }
term → *factor term′*	First(*term*) = { (, **number** }	Follow(*term*) = { $,), +, − }
term′ → *multop factor term′* | ε	First(*term′*) = { *, ε }	Follow(*term′*) = { $,), +, − }
mulop → *	First(*mulop*) = { * }	Follow(*mulop*) = { (, **number** }
factor → (*exp*) | **number**	First(*factor*) = { (, **number** }	Follow(*factor*) = { $,), +, −, * }

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) Expression Grammar

Parsing Stack	Input	Action
$ exp	(2 + *) $	exp→ term exp′
$ exp′ term	(2 + *) $	term → factor term′
$ exp′ term′ factor	(2 + *) $	factor → (exp)
$ exp′ term′) exp (	(2 + *) $	match
$ exp′ term′) exp	2 + *) $	exp → term exp′
$ exp′ term′) exp′ term	2 + *) $	term → factor term′
$ exp′ term′) exp′ term′ factor	2 + *) $	factor → 2
$ exp′ term′) exp′ term′ 2	2 + *) $	match
$ exp′ term′) exp′ term′	+ *) $	term′ → ε
$ exp′ term′) exp′	+ *) $	exp′ → addop term exp′
$ exp′ term′) exp′ term addop	+ *) $	addop → +
$ exp′ term′) exp′ term +	+ *) $	match
$ exp′ term′) exp′ term	*) $	scan
$ exp′ term′) exp′ term	) $	pop
$ exp′ term′) exp′	) $	exp′ → ε
$ exp′ term′)	) $	match
$ exp′ term′	$	term′ → ε
$ exp′	$	exp′ → ε
$	$	**accept**

Conclusions

- Top-down parsing
- Recursive descendant parsing
- LL(1) parser generation algorithm
- Right recursive and left-factored grammars
- Panic mode error recovery