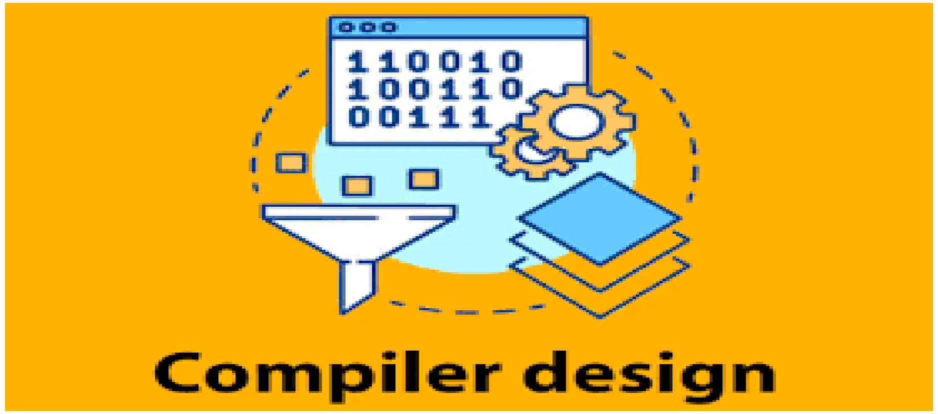
Menoufia University
Faculty of computers & Information
Computer Science Department.

Compiler Design

4 Year – first Semester

Lecture 6



DR. Eman Meslhy Mohamed
Lecturer at Computer Science department
2023-2024

Outline

- **LL(1) Grammar.**
- Pushdown Machine for LL(1) Grammar.
- Recursive Descent Parsers for LL(1) Grammar.
- **Translation Grammar**
- Pushdown Machine for translation Grammar.
- Recursive Descent Parsers for translation Grammar.

Quiz

- Find the selection sets for the following grammar. Is the grammar quasi-simple? If so, show a pushdown machine and a recursive descent parser (show methods S() and A() only) corresponding to this grammar.
- 1. $S \rightarrow b\ A\ b$
- 2. $S \rightarrow a$
- 3. $A \rightarrow \epsilon$
- 4. $A \rightarrow a\ S\ a$

Answer

- Sel(1) = {b}
- Sel(2) = {a}
- Sel(3) = FOL(A) = {b}
- Sel(4) = {a}

	a	b	↲
S	Rep (a) Retain	Rep (bAb) Retain	Reject
A	Rep (aSa) Retain	Pop Retain	Reject
a	Pop Advance	Reject	Reject
b	Reject	Pop Advance	Reject
∇	Reject	Reject	Accept

S
∇

Initial Stack

```
void S()
   {  if (inp=='b')                         // apply rule 1
         {  getInp();
            A();
            if (inp=='b') getInp();
            else Reject();
         }                                  // end rule 1
      else if (inp=='a') getInp();          // apply rule 2
            else reject();
   }
```

```
void A()
   {  if (inp=='b')   ;                     // apply rule 3
      else if (inp=='a') getInp()           // apply rule 4
         {  getInp();
            S();
            if (inp=='a') getInp();
            else reject();
         }                                  // end rule 4
      else reject();
   }
```

LL(1) Grammars

- Grammars that can be parsed top down by allowing rules of the form $N \rightarrow a$ where a is any string of terminals and **nonterminals**.
- The name LL(1) is left-most derivation when scanning the input from left to right if it can look ahead no more than one input symbol.

$S \rightarrow ABc$

$A \rightarrow bA$

$A \rightarrow \epsilon$

$B \rightarrow c$

Algorithm to find selection sets

1. Find nullable rules and nullable nonterminals.
2. Find Begins Directly With relation (BDW).
3. Find Begins With relation (BW).
4. Find First(x) for each symbol, x.
5. Find First(n) for the right side of each rule, n.
6. Find Followed Directly By relation (FDB).
7. Find Is Direct End Of relation (DEO).
8. Find Is End Of relation (EO).
9. Find Is Followed By relation (FB).
10. Extend FB to include endmarker.
11. Find Follow Set, Fol(A), for each nullable nonterminal, A.
12. Find Selection Set, Sel(n), for each rule, n.

Step 1. Find all nullable rules and nullable nonterminals:

- All ϵ rules are nullable rules.
- The nonterminal defined in a nullable rule is a nullable nonterminal.
- All rules in the form $A \rightarrow BCD\ldots$
- where B, C, D, ... are all nullable non-terminals, are nullable rules; the nonterminals defined by these rules are also nullable non-terminals.

$S \rightarrow ABc$
$A \rightarrow bA$
$A \rightarrow \epsilon$
$B \rightarrow c$

Nullable rules: rule 3
Nullable nonterminals: A

Step 2. Compute the relation Begins Directly With for each nonterminal:

- A BDW X if there is a rule $A \rightarrow \alpha X \beta$ such that:
  - α is a nullable string (a string of nullable nonterminals).
  - A represents a nonterminal and X represents a terminal or nonterminal.
  - β represents any string of terminals and nonterminals.

$S \rightarrow ABc$
$A \rightarrow bA$
$A \rightarrow \epsilon$
$B \rightarrow c$

S BDW A (from rule 1)
S BDW B (also from rule 1, because A is nullable)
A BDW b (from rule 2)
B BDW c (from rule 4)

Step 3. Compute the relation Begins With:

- BW is the reflexive transitive closure of BDW.
- In addition, BW should contain pairs of the form a BW a for each terminal a in the grammar.

S BDW A
S BDW B
A BDW b
B BDW c

```
S  BW  A
S  BW  B        (from BDW)
A  BW  b
B  BW  c

S  BW  b        (transitive)
S  BW  c

S  BW  S
A  BW  A
B  BW  B        (reflexive)
b  BW  b
c  BW  c
```

Step 4. Compute First(x) for each symbol x in the grammar.

- First(A) = set of all terminals b, such that A BW b for each nonterminal A.
- First(t) = {t} for each terminal t.

$S \to ABc$
$A \to bA$
$A \to \epsilon$
$B \to c$

```
S BW A
S BW B        (from BDW)
A BW b
B BW c

S BW b        (transitive)
S BW c

S BW S
A BW A
B BW B        (reflexive)
b BW b
c BW c
```

First(S) = {b,c}
First(A) = {b}
First(B) = {c}
First(b) = {b}
First(c) = {c}

Step 5. Compute First of right side of each rule:

- First (XYZ...) = First(X)
  U First(Y) if X is nullable
  U First(Z) if Y is also nullable

$S \rightarrow ABc$
$A \rightarrow bA$
$A \rightarrow \epsilon$
$B \rightarrow c$

First(S) = {b,c}
First(A) = {b}
First(B) = {c}
First(b) = {b}
First(c) = {c}

If the grammar contains no nullable rules, you may skip to step 12 at this point.

1. First(ABc) = First(A) U First(B) = {b,c} (because A is nullable)
2. First(bA) = {b}
3. First(ϵ) = {}
4. First(c) = {c}

Step 6. Compute the relation Is Followed Directly By:

- B FDB X
- if there is a rule of the form $A \rightarrow \alpha B \beta X \gamma$
- where β is a string of nullable nonterminals, α, γ are strings of symbols, X is any symbol, and A and B are nonterminals.

$S \rightarrow ABc$
$A \rightarrow bA$
$A \rightarrow \epsilon$
$B \rightarrow c$

A FDB B	(from rule 1)
B FDB c	(from rule 1)

Step 7. Compute the relation Is Direct End Of :

- X DEO A
- if there is a rule of the form: $A \rightarrow \alpha X \beta$
- where β is a string of nullable nonterminals, α is a string of symbols, and X is a single grammar symbol.

$S \rightarrow ABc$
$A \rightarrow bA$
$A \rightarrow \epsilon$
$B \rightarrow c$

c DEO S	(from rule 1)
A DEO A	(from rule 2)
b DEO A	(from rule 2, since A is nullable)
c DEO B	(from rule 4)

Step 8. Compute the relation Is End Of :

- X EO Y
- EO is the reflexive transitive closure of DEO.

c DEO S
A DEO A
b DEO A
c DEO B

c EO S
A EO A (from DEO)
b EO A
c EO B

(no transitive entries)

c EO c
S EO S (reflexive)
b EO b
B EO B

Step 9. Compute the relation Is Followed By:

- W FB Z
- If W EO X and X FDB Y and Y BW Z
- then W FB Z

A EO A	A FDB B	B BW B	A FB B
		B BW c	A FB c
b EO A		B BW B	b FB B
		B BW c	b FB c
B EO B	B FDB c	c BW c	B FB c
c EO B		c BW c	c FB c

Step 10. Extend the FB relation to include endmarker:

- A FB ← if A EO S where A represents any nonterminal and S represents the starting nonterminal.

c EO S
A EO A (from DEO)
b EO A
c EO B

(no transitive entries)

c EO c
S EO S (reflexive)
b EO b
B EO B

S FB ← because S EO S

Step 11. Compute the Follow Set for each nullable nonterminal:

- Fol(A) = {t: A FB t}

- Fol(A) = {c} since A is the only nullable nonterminal and A FB c.

A	FB	B
A	FB	c
b	FB	B
b	FB	c
B	FB	c
c	FB	c

$S \rightarrow ABc$
$A \rightarrow bA$
$A \rightarrow \epsilon$
$B \rightarrow c$

Step 12. Compute the Selection Set for each rule:

- i. $A \rightarrow \alpha$
- if rule i is not a nullable rule, then Sel(i) = First(α)
- if rule i is a nullable rule, then Sel(i) = First(α) U Fol(A)

$S \rightarrow ABc$
$A \rightarrow bA$
$A \rightarrow \epsilon$
$B \rightarrow c$

Sel(1) = First(ABc) = {b,c}
Sel(2) = First(bA) = {b}
Sel(3) = First(ε) U Fol(A) = {} U {c} = {c}
Sel(4) = First(c) = {c}

- A context-free grammar is LL(1) if rules defining the same nonterminal always have disjoint selection sets.

$S \rightarrow ABc$

$A \rightarrow bA$

$A \rightarrow \epsilon$

$B \rightarrow c$

This Grammar is LL(1) ???

Sel(1) = First(ABc) = {b,c}

Sel(2) = First(bA) = {b}

Sel(3) = First(ε) U Fol(A) = {} U {c} = {c}

Sel(4) = First(c) = {c}

Pushdown Machines for LL(1) Grammars

- For a rule in the grammar, $A \rightarrow \alpha$, fill in the cells in the row for nonterminal A and in the columns for the selection set of that rule with Rep(α^r), Retain.
- For ϵ rules, fill in Pop, Retain in the columns for the selection set.
- For each terminal symbol, enter Pop, Advance in the cell in the row and column labeled with that terminal.
- The cell in the row labeled $\triangledown$ and the column labeled $\leftarrow$ should contain *Accept*.
- *All* other cells are Reject.

Pushdown Machines for LL(1) Grammars

$S \rightarrow ABc$
$A \rightarrow bA$
$A \rightarrow \epsilon$
$B \rightarrow c$

Sel(1) = {b,c}
Sel(2) = {b}
Sel(3) = {c}
Sel(4) = {c}

	b	c	↩
S	Rep (cBA) Retain	Rep (cBA) Retain	Reject
A	Rep (Ab) Retain	Pop Retain	Reject
B	Reject	Rep (c) Retain	Reject
b	Pop Advance	Reject	Reject
c	Reject	Pop Advance	Reject
∇	Reject	Reject	Accept

S
∇

Initial Stack

Recursive Descent for LL(1) Grammars

$S \rightarrow ABc$
$A \rightarrow bA$
$A \rightarrow \epsilon$
$B \rightarrow c$

Sel(1) = {b,c}
Sel(2) = {b}
Sel(3) = {c}
Sel(4) = {c}

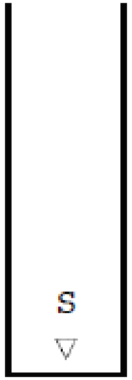
```
void S ()
    {     if (inp=='b' || inp=='c')
              {     A ();
                    B ();
                    if (inp=='c') getInp();
                    else reject();
              }
          else reject();
    }
```

```
void A ()
    {     if (inp=='b')
              {     getInp();
                    A ();
              }
          else if (inp=='c') ;
          else reject();
    }
```

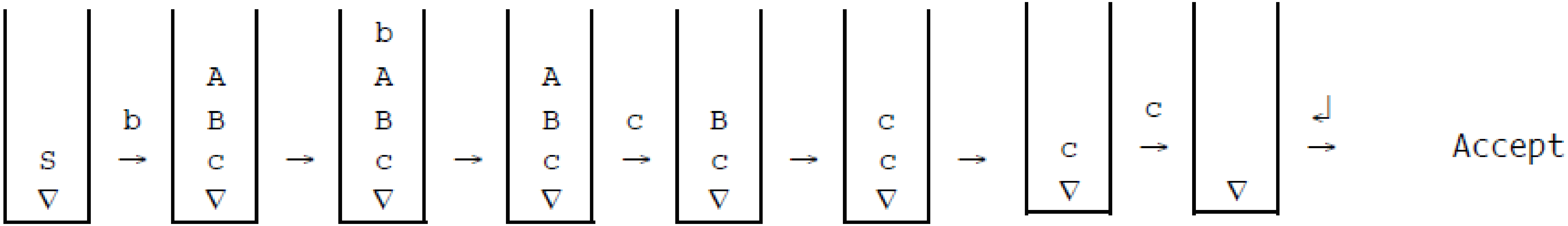
```
void B ()
    {     if (inp=='c') getInp();
    else reject();
    }
```

- Show the sequence of stacks that occurs when the pushdown machine parses the string bcc←

	b	c	↩
S	Rep (cBA) Retain	Rep (cBA) Retain	Reject
A	Rep (Ab) Retain	Pop Retain	Reject
B	Reject	Rep (c) Retain	Reject
b	Pop Advance	Reject	Reject
c	Reject	Pop Advance	Reject
▽	Reject	Reject	Accept



Initial Stack



Parsing Arithmetic Expressions Top Down

- We wish to determine whether this grammar is LL(1).

1. $Expr \rightarrow Expr + Term$
2. $Expr \rightarrow Term$
3. $Term \rightarrow Term * Factor$
4. $Term \rightarrow Factor$
5. $Factor \rightarrow (Expr)$
6. $Factor \rightarrow var$

1. Expr → Expr + T
2. Expr → T erm
3. T erm → T erm∗
4. T erm → Factor
5. Factor → (Expr)
6. Factor → var

Nullable rules: non
Nullable nontermin

(,var}
(,var}
{(,var}

Term) = {(,var}
{(,var}
Factor) = {(,var}
= {(,var}
)) = {(}
var}

IS This grammar is LL(1) ??

Expr B
Expr B
Term B
Term B
Factor
Factor

No, Why?

left recursion

- Expr $\rightarrow$ Expr + Term
- Term $\rightarrow$ Term $*$ Factor
- Any grammar with left recursion cannot be LL(1).

- The left recursion can be eliminated by rewriting the grammar with an equivalent grammar that does not have left recursion.

$$A \rightarrow A\alpha$$
$$A \rightarrow \beta$$

$$A \rightarrow \beta R$$
$$R \rightarrow \alpha R$$
$$R \rightarrow \epsilon$$

left recursion

A → βR

→ αR

→ ϵ

IS the modified grammar is LL(1) ????

1. Expr → Term Elist
2. Elist → + Term Elist
3. Elist → ϵ
4. Term → Factor Tlist
5. Tlist → * Factor Tlist
6. Tlist → ϵ
7. Factor → (Expr)
8. Factor → var

Translation Grammar

- The programs we have developed can check only for syntax errors; they cannot produce output.
- For this purpose, we now introduce ***action symbols*** which are intended to give us the capability of producing output.
- A grammar containing action symbols is called a ***translation grammar***. $A \rightarrow \alpha \{action\} \beta$
- To find the selection sets in a translation grammar, simply remove all the action symbols, This results in what is called the ***underlying grammar***.

Example

- An example of a translation grammar to translate infix expressions to postfix form involving addition and multiplication:

Grammar without Action Symbols

1. Expr → Term Elist
2. Elist → + Term Elist
3. Elist → ε
4. Term → Factor Tlist
5. Tlist → * Factor Tlist
6. Tlist → ε
7. Factor → (Expr)
8. Factor → var

Grammar with Action Symbols

G17:

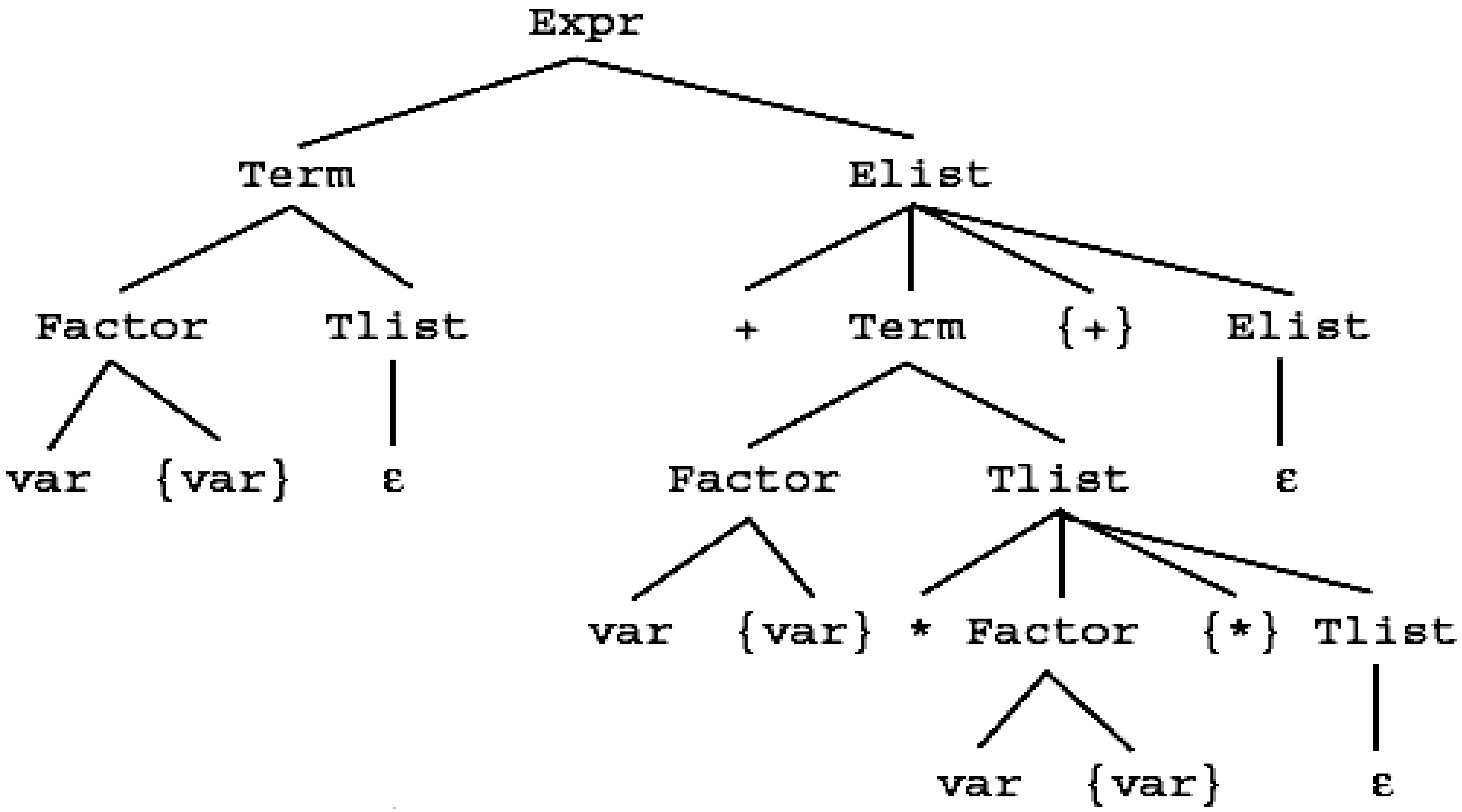
1. **Expr → Term Elist**
2. **Elist → + Term {+} Elist**
3. **Elist → ε**
4. **Term → Factor Tlist**
5. **Tlist → * Factor {*} Tlist**
6. **Tlist → ε**
7. **Factor → (Expr)**
8. **Factor → var {var}**

Example

Show the derivation tree for the expression var + var ∗ var

G17:

1. Expr $\rightarrow$ Term Elist
2. Elist $\rightarrow$ + Term {+} Elist
3. Elist $\rightarrow$ ε
4. Term $\rightarrow$ Factor Tlist
5. Tlist $\rightarrow$ ∗ Factor {∗} Tlist
6. Tlist $\rightarrow$ ε
7. Factor $\rightarrow$ (Expr)
8. Factor $\rightarrow$ var {var}



- Separating out the action symbols gives the output defined by the translation grammar:

{var} {var} {var} {∗ } {+}

Implementing Translation Grammars with Pushdown Translators

- To implement a translation grammar with a pushdown machine,
- action symbols should be treated as stack symbols and are pushed onto the stack in exactly the same way as terminals and nonterminals occurring on the right side of a grammar rule.

- In addition, each action symbol {*A*} representing output should label a row of the pushdown machine table.

- Every column of that row should contain the entry

  Pop, Retain, Out(*A*).

Example

- Show an extended pushdown translator for the following translation grammar

1. $S \rightarrow \{print\}aS$
2. $S \rightarrow bB$
3. $B \rightarrow \{print\}$

- selection sets:

Sel(1) = {a}
Sel(2) = {b}
Sel(3) = {←}

	a	b	↩
S	Rep (Sa{print}) Retain	Rep (Bv) Retain	Reject
B	Reject	Reject	Rep ({print}) Retain
a	Pop Adv		
b	Reject	Pop Adv	
{print}	Pop Retain Out ({print})	Pop Retain Out ({print})	Pop Retain Out ({print})
▽	Reject	Reject	Accept

S
▽

Initial Stack

Example

- Show an extended pushdown translator for the infix to postfix translator constructed from grammar

G17:

1. Expr $\rightarrow$ Term Elist
2. Elist $\rightarrow$ + Term {+} Elist
3. Elist $\rightarrow$ ε
4. Term $\rightarrow$ Factor Tlist
5. Tlist $\rightarrow$ $*$ Factor {$*$} Tlist
6. Tlist $\rightarrow$ ε
7. Factor $\rightarrow$ (Expr)
8. Factor $\rightarrow$ var {var}

Sel(1) = First(Term Elist) = {(,var}
Sel(2) = First(+ Term Elist) = { +}
Sel(3) = Fol(Elist) = {),←}
Sel(4) = First(Factor Tlist) = {(,var}
Sel(5) = First(* Factor Tlist) = {*}
Sel(6) = Fol(Tlist) = {+,),←}
Sel(7) = First((Expr)) = {(}
Sel(8) = First(var) = {var}

	var	+	*	(	)	↩
Expr	Rep(Elist Term) Retain			Rep(Elist Term) Retain		
Elist		Rep(Elist {+}Term+) Retain			Pop Retain	Pop Retain
Term	Rep(Tlist Factor) Retain			Rep(Tlist Factor) Retain		
Tlist		Pop Retain	Rep(Tlist {*}Factor*) Retain		Pop Retain	Pop Retain
Factor	Rep({var} var) Retain			Rep( )Expr( ) Retain		
var	Pop Advance					
+		Pop Advance				
*			Pop Advance			
(				Pop Advance		
)					Pop Advance	
{var}	Pop Retain Out (var)	Pop Retain Out (var)	Pop Retain Out (var)	Pop Retain Out (var)	Pop Retain Out (var)	Pop Retain Out (var)
{+}	Pop Retain Out (+)	Pop Retain Out (+)	Pop Retain Out (+)	Pop Retain Out (+)	Pop Retain Out (+)	Pop Retain Out (+)
{*}	Pop Retain Out (*)	Pop Retain Out (*)	Pop Retain Out (*)	Pop Retain Out (*)	Pop Retain Out (*)	Pop Retain Out (*)
▽						Accept

Expr
▽

Initial Stack

Implementing Translation Grammars with Recursive Descent

- Show the Recursive Descent translator for the following translation grammar

1. $S \rightarrow \{print\}aS$
2. $S \rightarrow bB$
3. $B \rightarrow \{print\}$

- selection sets:

Sel(1) = {a}

Sel(2) = {b}

Sel(3) = {←}

```
void S ()
{  if (inp=='a')
    {      getInp();                       // apply rule 1
      System.out.println ("print");
      S();
    }                                     // end rule 1
   else if (inp=='b')
   {        getInp();                     // apply rule 2
      B();
   }                                      // end rule 2
   else Reject ();
}

void B ()
{  if (inp==Endmarker) System.out.println ("print");
                                   // apply rule 3
   else Reject ();
}
```

Example

- Show the Recursive Descent translator for the infix to postfix translator constructed from grammar

G17:

1. Expr → Term Elist
2. Elist → + Term {+} Elist
3. Elist → ε
4. Term → Factor Tlist
5. Tlist → ∗ Factor {∗} Tlist
6. Tlist → ε
7. Factor → (Expr)
8. Factor → var {var}

Sel(1) = First(Term Elist) = {(,var}
Sel(2) = First(+ Term Elist) = { +}
Sel(3) = Fol(Elist) = {),←}
Sel(4) = First(Factor Tlist) = {(,var}
Sel(5) = First(* Factor Tlist) = {*}
Sel(6) = Fol(Tlist) = {+,),←}
Sel(7) = First((Expr)) = {(}
Sel(8) = First(var) = {var}

```
void Expr ()
{  if (inp=='(' || inp==var)
   {     Term ();                          // apply rule 1
      Elist ();
   }                                       // end rule 1
      else Reject ();
}
```

```
void Elist ()
{  if (inp=='+')
   {  getInp();                          // apply rule 2
      Term ();
      System.out.println ('+');
      Elist ();
   }                                     // end rule 2
   else if (inp==Endmarker || inp==')') ;     // apply rule 3
   else Reject ();
}
```

Example

- Show the Recursive Descent translator for the infix to postfix translator constructed from grammar

G17:

1. Expr $\rightarrow$ Term Elist
2. Elist $\rightarrow$ + Term {+} Elist
3. Elist $\rightarrow$ ε
4. Term $\rightarrow$ Factor Tlist
5. Tlist $\rightarrow$ $*$ Factor {$*$} Tlist
6. Tlist $\rightarrow$ ε
7. Factor $\rightarrow$ (Expr)
8. Factor $\rightarrow$ var {var}

Sel(1) = First(Term Elist) = {(,var}
Sel(2) = First(+ Term Elist) = { +}
Sel(3) = Fol(Elist) = {),←}
Sel(4) = First(Factor Tlist) = {(,var}
Sel(5) = First(* Factor Tlist) = {*}
Sel(6) = Fol(Tlist) = {+,),←}
Sel(7) = First((Expr)) = {(}
Sel(8) = First(var) = {var}

```
void Term ()
{  if (inp=='(' || inp==var)
   {  Factor ();                        // apply rule 4
      Tlist ();
   }                                    // end rule 4
      else Reject ();
}

void Tlist ()
{  if (inp=='*')
   {  getInp();                         // apply rule 5
      Factor ();
      System.out.println ('*');
      Tlist ();
   }                                    // end rule 5
      else if (inp=='+' || inp==')' || inp=Endmarker) ;
                                        // apply rule 6
      else Reject ();
}
```

Example

- Show the Recursive Descent translator for the infix to postfix translator constructed from grammar

G17:

1. Expr $\rightarrow$ Term Elist
2. Elist $\rightarrow$ + Term {+} Elist
3. Elist $\rightarrow$ ε
4. Term $\rightarrow$ Factor Tlist
5. Tlist $\rightarrow$ $*$ Factor {$*$} Tlist
6. Tlist $\rightarrow$ ε
7. Factor $\rightarrow$ (Expr)
8. Factor $\rightarrow$ var {var}

Sel(1) = First(Term Elist) = {(,var}
Sel(2) = First(+ Term Elist) = { +}
Sel(3) = Fol(Elist) = {),←}
Sel(4) = First(Factor Tlist) = {(,var}
Sel(5) = First(* Factor Tlist) = {*}
Sel(6) = Fol(Tlist) = {+,),←}
Sel(7) = First((Expr)) = {(}
Sel(8) = First(var) = {var}

```
void Factor ()
{  if (inp=='(')
   {  getInp();                          // apply rule 7
      Expr ();
      if (inp==')') getInp();
      else Reject ();
   }                                     // end rule 7
   else if (inp==var)
   {  getInp();                          // apply rule 8
      System.out.println ("var");
   }                                     // end rule 8
   else Reject ();
}
```

Thanks