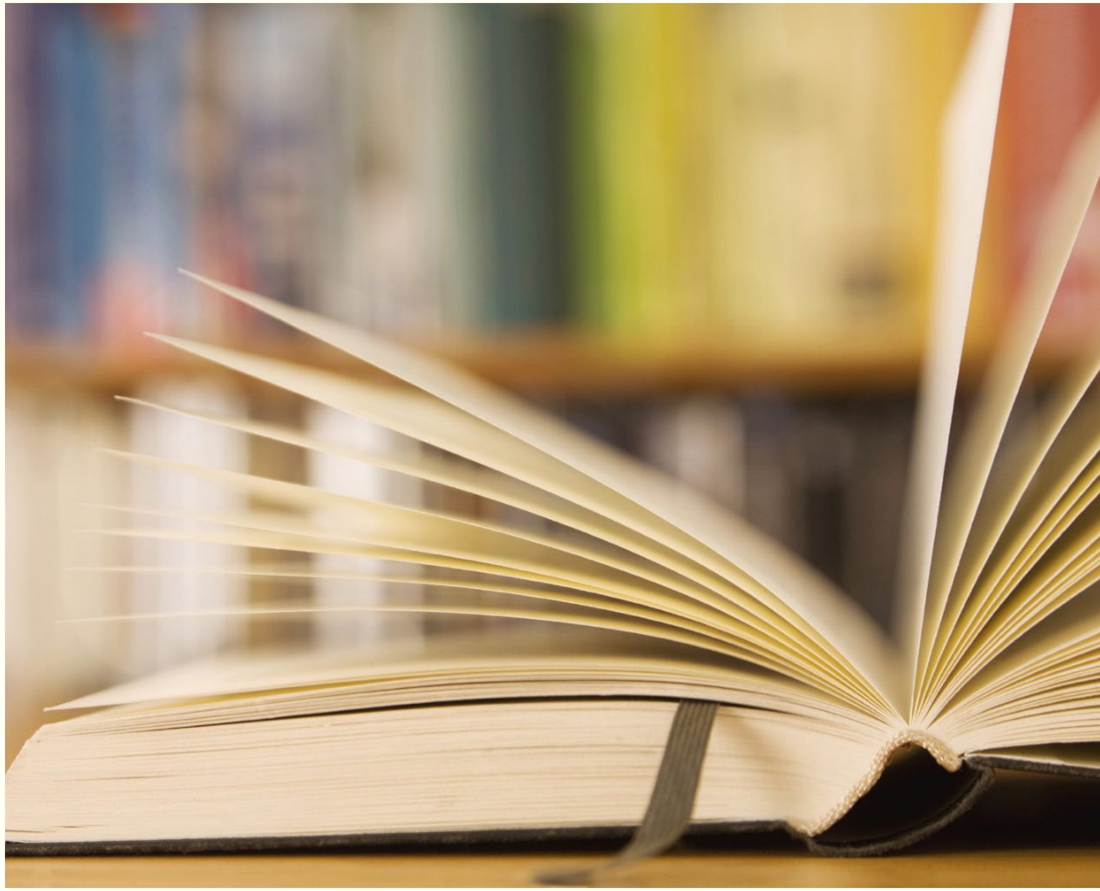
SYNTAX AND SEMANTIC ANALYSIS

Tajul Rosli Razak, PhD



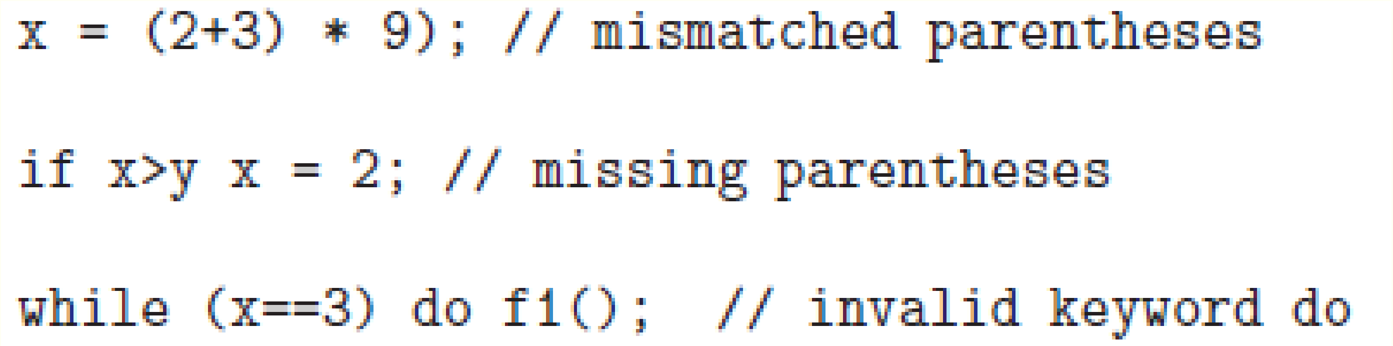
Source : Compiler Design: Theory, Tools, and Examples
Seth D. Bergmann
Rowan University

CONTENTS

GRAMMARS, LANGUAGES, AND PUSHDOWN MACHINES

Syntax Analysis

- Is the second phase of compiler.
- Input: Stream of tokens from lexical analysis phase
- Output: Stream of atoms or syntax tree.
- The purpose is to detect and scan for any syntax errors.



```
x = (2+3) * 9); // mismatched parentheses

if x>y x = 2; // missing parentheses

while (x==3) do f1();  // invalid keyword do
```

Grammar

- A grammar is a list of rules which can be used to produce or generate all the strings of a language, and which does not generate any strings which are not in the language.

- a grammar consists of:

1. A finite set of characters, called the input alphabet, the input symbols, or terminal symbols.

2. A finite set of symbols, distinct from the terminal symbols, called nonterminal symbols, exactly one of which is designated the starting nonterminal (if no nonterminal is explicitly designated as the starting nonterminal, it is assumed to be the nonterminal defined in the first rule).

3. A finite list of rewriting rules, also called productions, which define how strings in the language may be generated. Each of these rewriting rules is of the form a → b, where a and b are arbitrary strings of terminals and nonterminals, and a is not null.

Example G1

- The grammar G1, which consists of four rules, the terminal symbols {0,1}, and the starting nonterminal, S.
- G1:

  1. S → 0S0
  2. S → 1S1
  3. S → 0
  4. S → 1
- An example of a derivation using this grammar is:
- S ⇒ 0S0 ⇒ 00S00 ⇒ 001S100⇒ 0010100
- Thus, 0010100 is in L(G1)
- L(G1) = {0, 1, 000, 010, 101, 111, 00000, ... }

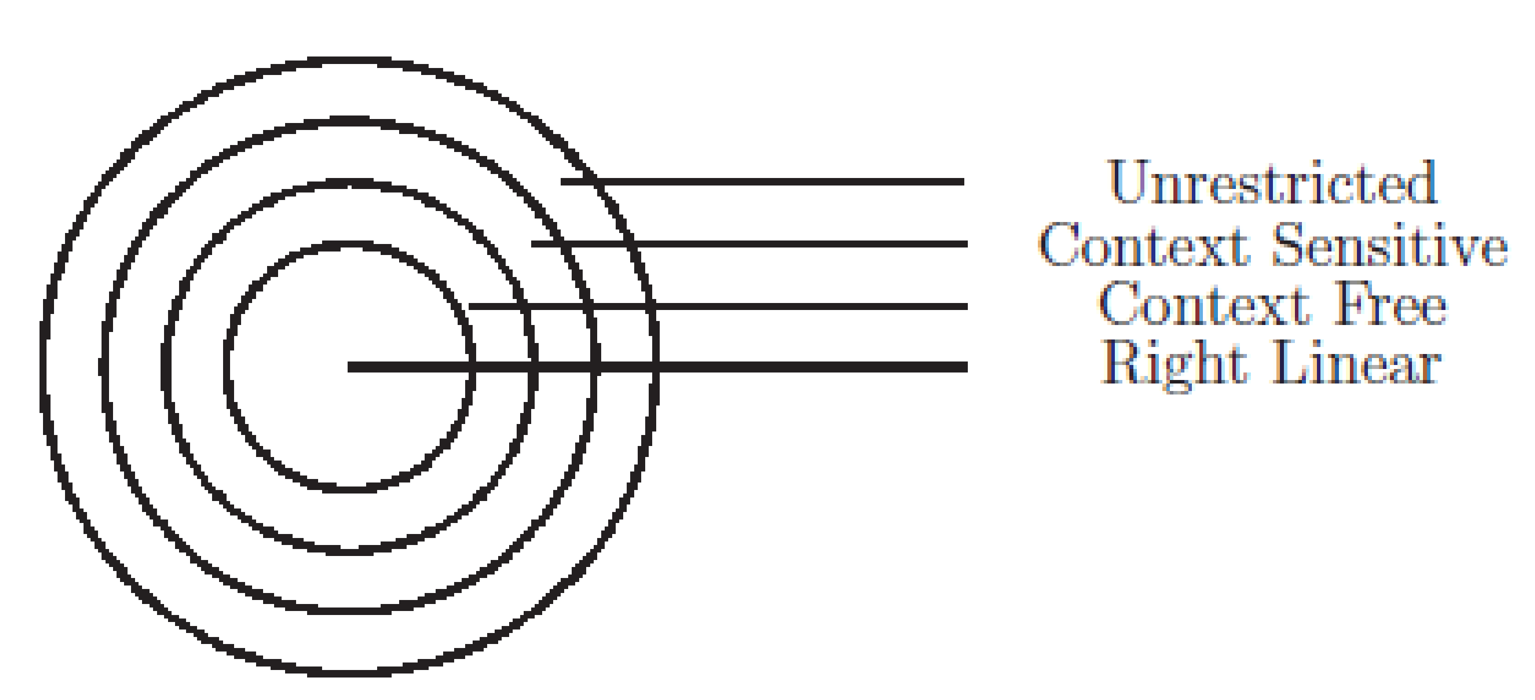
Example G2

- The terminal symbols are {a,b} (ε represents the null string and is not a terminal symbol).
- G2:
  - 1. S → ASB
  - 2. S → ε
  - 3. A → a
  - 4. B → b
- S ⇒ ASB ⇒ AASBB ⇒ AaSBB ⇒ AaBB ⇒ AaBb ⇒ Aabb ⇒ aabb
- Thus, aabb is in L(G2).
- L(G2) = {ε, ab, aabb, aaabbb, aaaabbbb, aaaaabbbbb, ...} = $\{a^nb^n\}$ such that $n \geq 0$
- This language is the set of all strings of a's and b's which consist of zero or more a's followed by exactly the same number of b's.

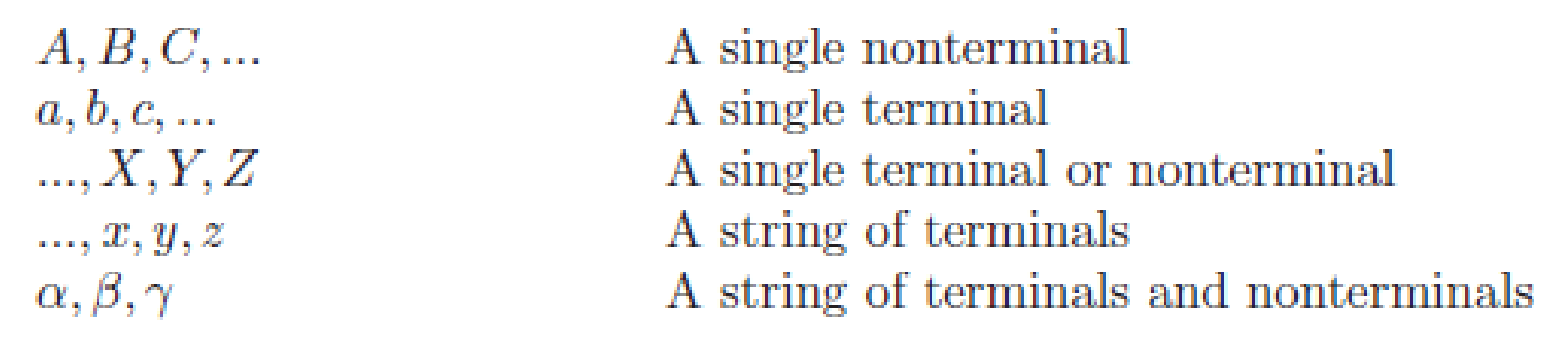
Equivalence

- Two grammars, g1 and g2, are said to be equivalent if L(g1) = L(g2)
- In this previous example string derivation from G1 is not the same with string derived from G2, thus L(G1) is not equal to L(G2)

Classes of Grammar (Chomsky's classification of grammars)



Symbol and meaning



$A, B, C, \ldots$	A single nonterminal
$a, b, c, \ldots$	A single terminal
$\ldots, X, Y, Z$	A single terminal or nonterminal
$\ldots, x, y, z$	A string of terminals
α, β, γ	A string of terminals and nonterminals

Explanation

- 0. Unrestricted: An unrestricted grammar is one in which there are no restrictions on the rewriting rules.

  example: SaB → cS

- 1. Context-Sensitive: A context-sensitive grammar is one in which each rule must be of the form:

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

where each of α, β, and γ is any string of terminals and nonterminals (including ε), and A represents a single nonterminal.

example: SaB → caB

continue

- 2. Context-Free: A context-free grammar is one in which each rule must be of the form:

  A → α

  where A represents a single nonterminal and α is any string of terminals and nonterminals.

  example : A → aABb

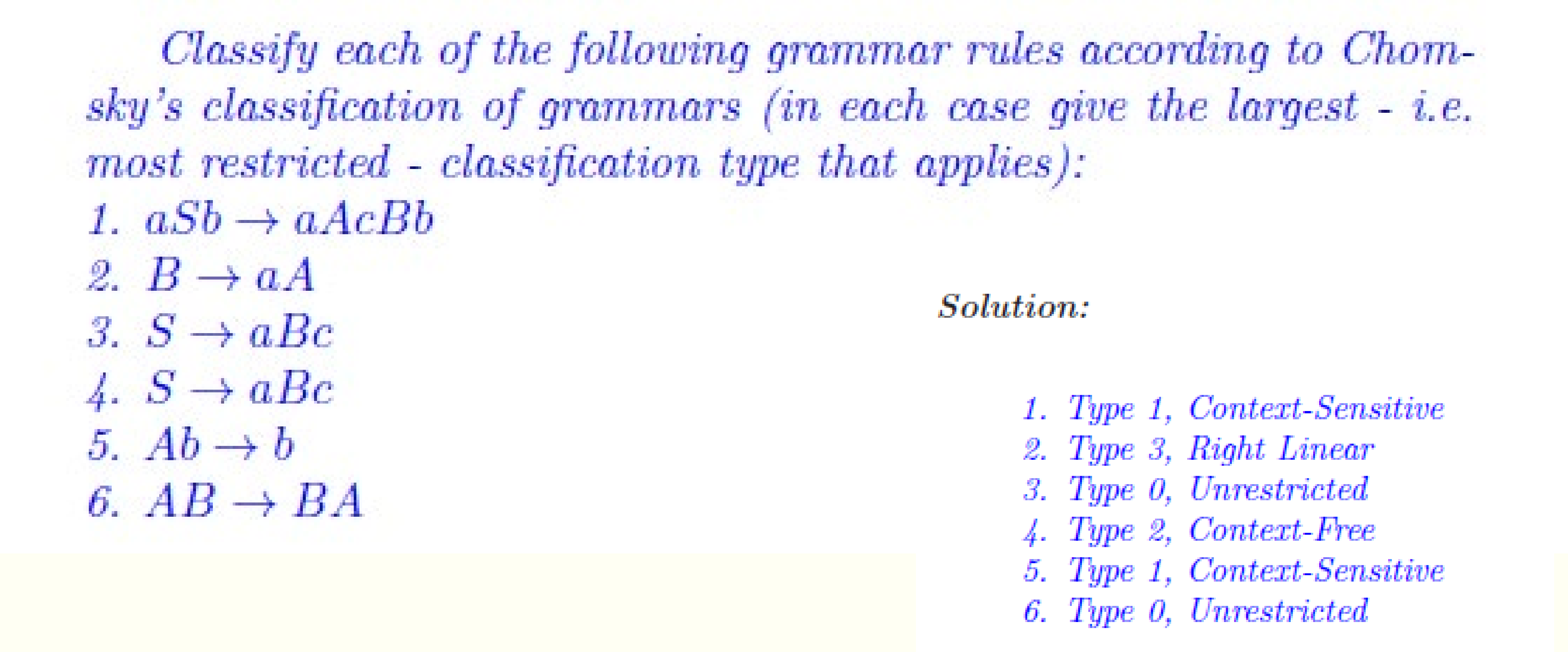
- 3. Right Linear: A right linear grammar is one in which each rule is of the form:

  A → aB or A → a

  where A and B represent nonterminals, and a represents a terminal.

  - used to define lexical items such as identifiers, constants, and keywords

Exercise



Classify each of the following grammar rules according to Chomsky's classification of grammars (in each case give the largest - i.e. most restricted - classification type that applies):

1. $aSb \rightarrow aAcBb$
2. $B \rightarrow aA$
3. $S \rightarrow aBc$
4. $S \rightarrow aBc$
5. $Ab \rightarrow b$
6. $AB \rightarrow BA$

Solution:

1. *Type 1, Context-Sensitive*
2. *Type 3, Right Linear*
3. *Type 0, Unrestricted*
4. *Type 2, Context-Free*
5. *Type 1, Context-Sensitive*
6. *Type 0, Unrestricted*

CONTEXT-FREE GRAMMAR

Overview

- programming languages are typically specified with context-free grammars.
- can be represented in a form called Backus-Naur Form (BNF)
  - nonterminals are enclosed in angle brackets <>, and
  - arrow is replaced by a ::=
  - For example:

Context Free grammar	BNF
$S \to aSb$	$\langle S \rangle ::= a \langle S \rangle b$
$S \to aSb$ $S \to \varepsilon$	$\langle S \rangle ::= a \langle S \rangle b \mid e$

Left/right most derivation

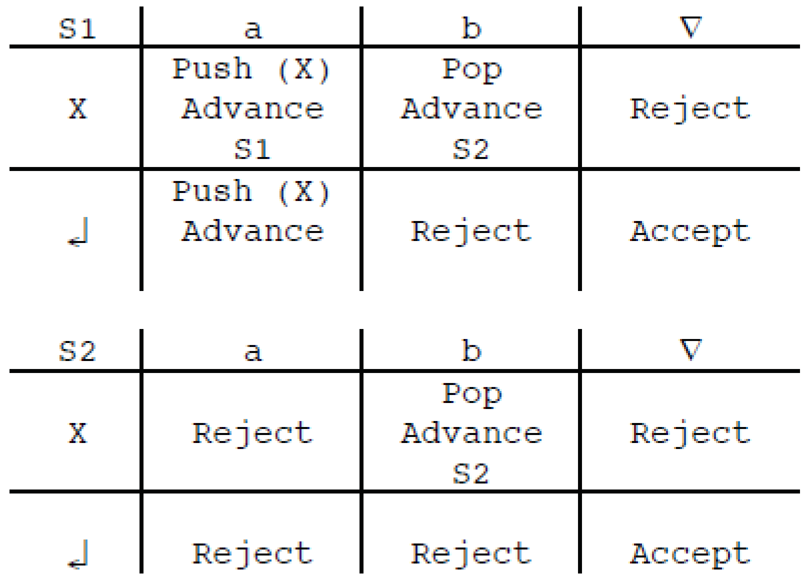
- A left-most derivation is one in which the left-most nonterminal is always the one to which a rule is applied.
- A right-most derivation is one in which the right-most nonterminal is always the one to which a rule is applied.
- Two special types of **linear grammars** are the following: the **left-linear** or **left-**regular **grammars**, in which all nonterminals in **right-**hand sides are at the **left** ends; the **right-linear** or **right**-regular **grammars**, in which all nonterminals in **right**-hand sides are at the **right** ends

PUSH-DOWN MACHINE

Introduction

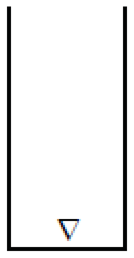
- A finite set of states, one of which is designated the starting state.
- A finite set of input symbols, the input alphabet.
- An infinite stack and a finite set of stack symbols which may be pushed on top or removed from the top of the stack in a last-in first-out manner.
- A state transition function
- the stack operations, push(X) or pop, where X is one of the stack symbols.
- A state transition may include an exit from the machine labeled either Accept or Reject.

Example: Machine accept the input string aabb

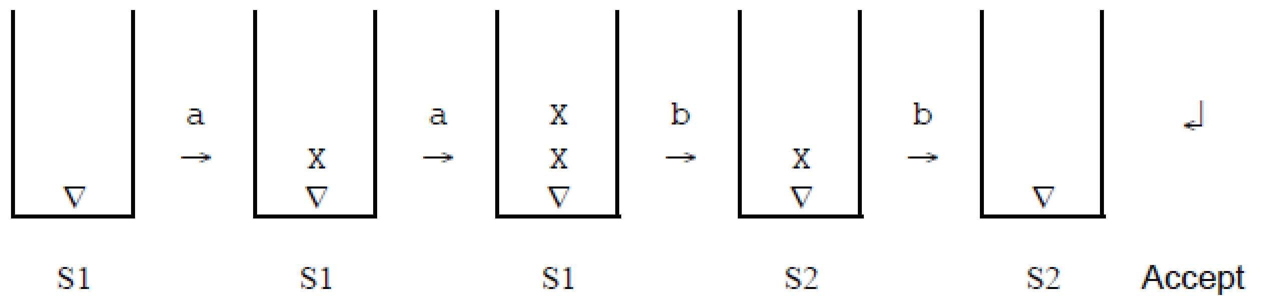


S1	a	b	∇
X	Push (X) Advance S1	Pop Advance S2	Reject
↲	Push (X) Advance	Reject	Accept

S2	a	b	∇
X	Reject	Pop Advance S2	Reject
↲	Reject	Reject	Accept



Initial Stack



AMBIGUITY IN PROGRAMMING

Ambiguous

- A context-free grammar is said to be ambiguous if there is more than one derivation tree for a particular string.

- For example:

G4:

1. Expr → Expr + Expr

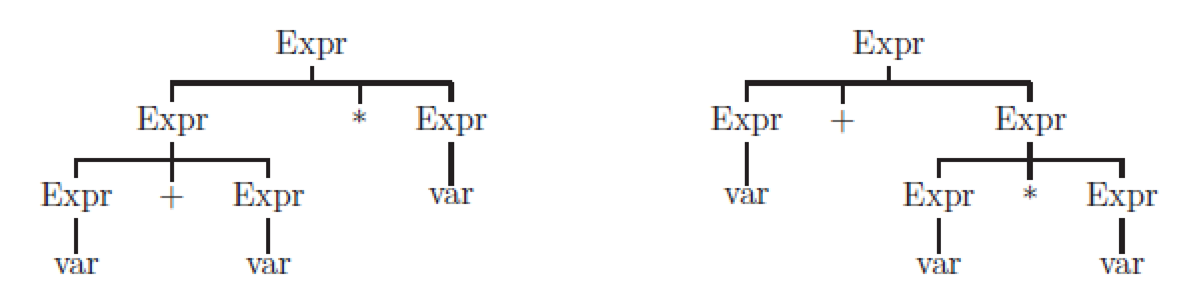
2. Expr → Expr * Expr

3. Expr → (Expr)

4. Expr → var

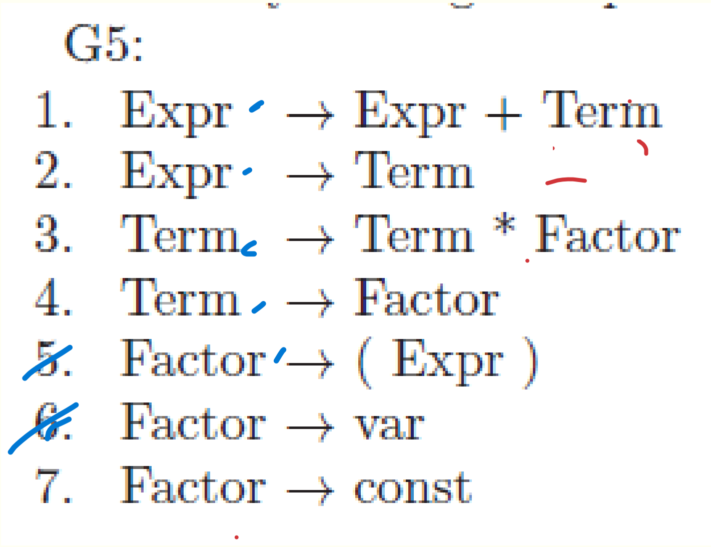
5. Expr → const

Two different derivation tree for string var + var * var



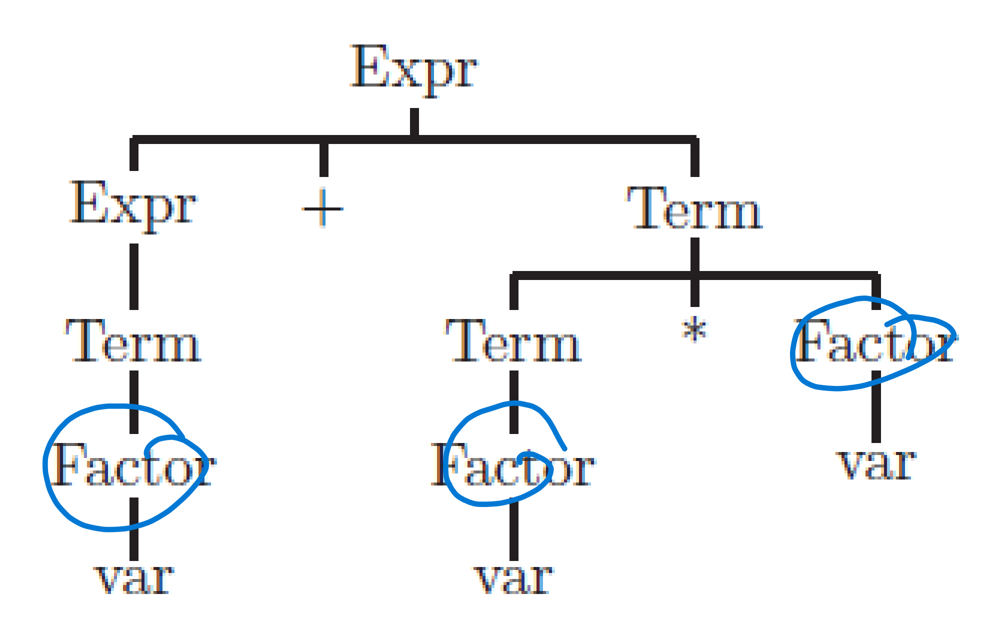
Elimination of Ambiguity using Precedence Rules

- Multiplication/division take precedence over addition/subtraction.
- Exponential take precedence over multiplication.

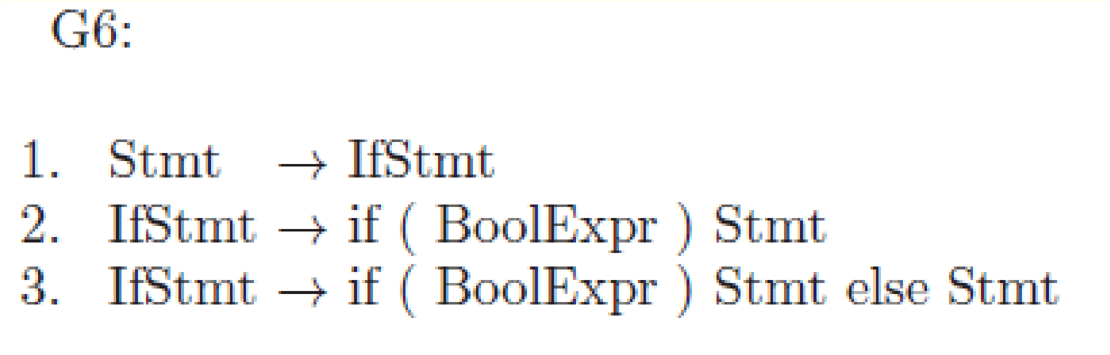


G5:

1. Expr → Expr + Term
2. Expr → Term
3. Term → Term * Factor
4. Term → Factor
~~5.~~ Factor → (Expr)
~~6.~~ Factor → var
7. Factor → const

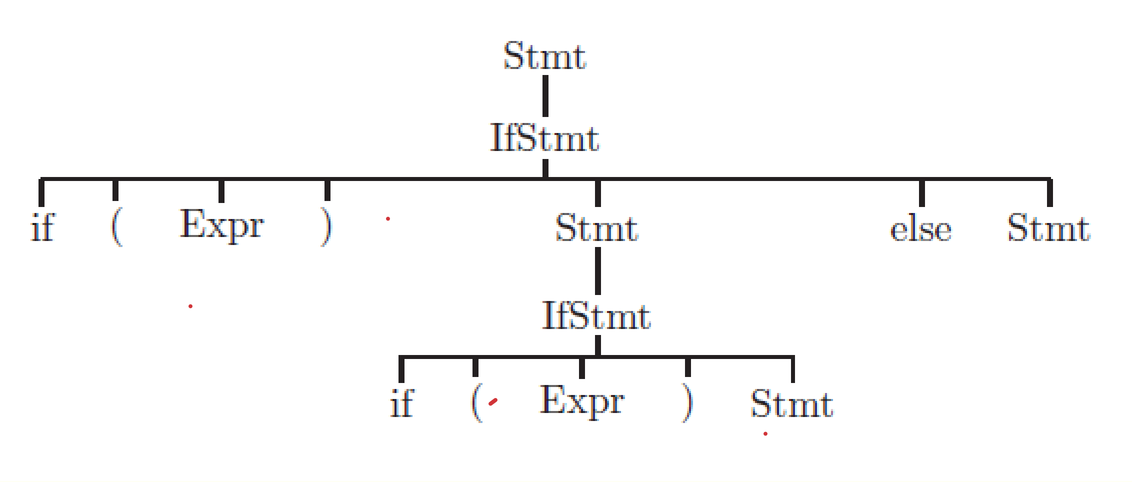


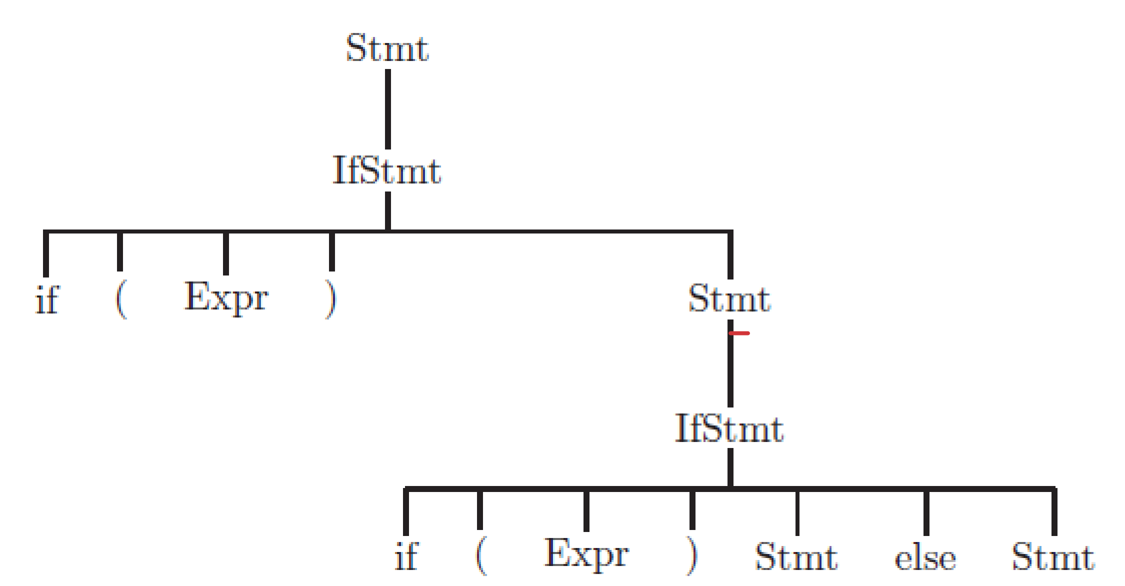
Another Example: If statement



G6:

1. Stmt → IfStmt
2. IfStmt → if (BoolExpr) Stmt
3. IfStmt → if (BoolExpr) Stmt else Stmt

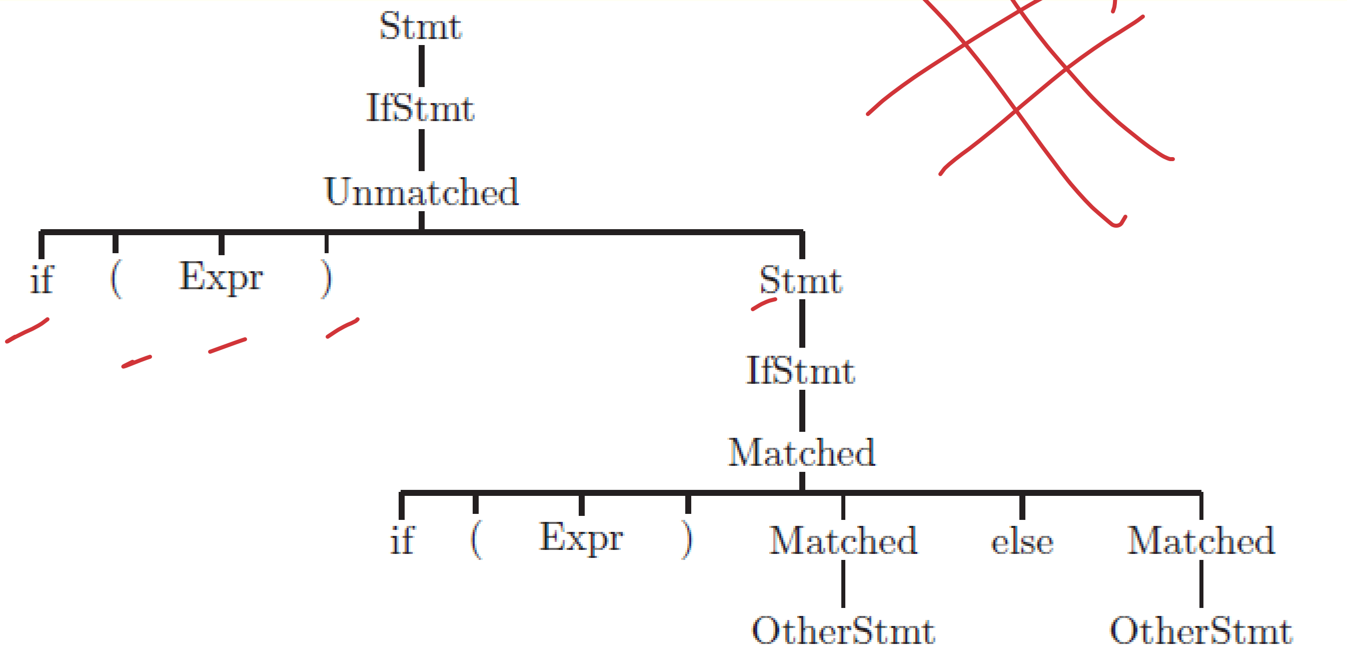




Elimination of Ambiguity

G7:

1. Stmt → IfStmt
2. IfStmt → Matched
3. IfStmt → Unmatched
4. Matched → if (BoolExpr) Matched else Matched
5. Matched → OtherStmt
6. Unmatched → if (BoolExpr) Stmt
7. Unmatched → if (BoolExpr) Matched else Unmatched



2

Isomorphic

Two grammars are said to be isomorphic if there is a one-to-one correspondence between the two grammars for every symbol of every rule. For example, the following two grammars are seen to be isomorphic, simply by making the following substitutions: substitute B for A, x for a, and y for b.

G1	G2
1. $S \to aAb$	1. $S \to xBy$ a A b
2. $A \to bAa$	2. $B \to yBx$
3. $A \to a$	3. $B \to x$

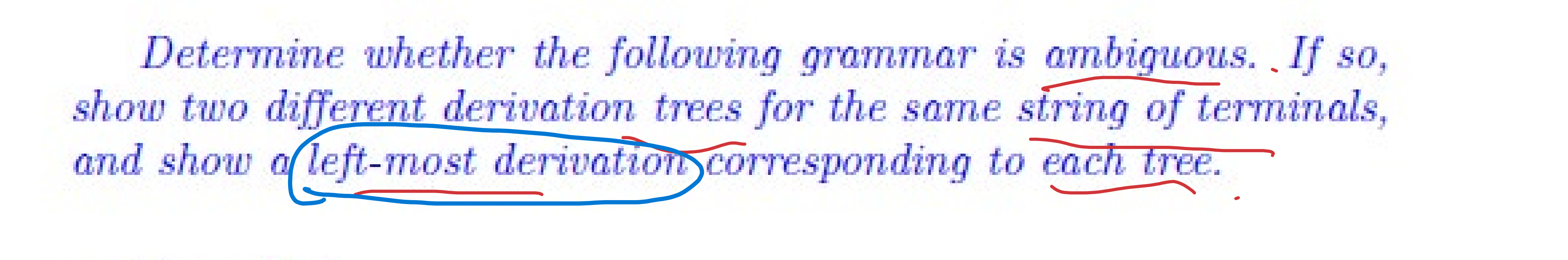
B → A

x → a

y → b

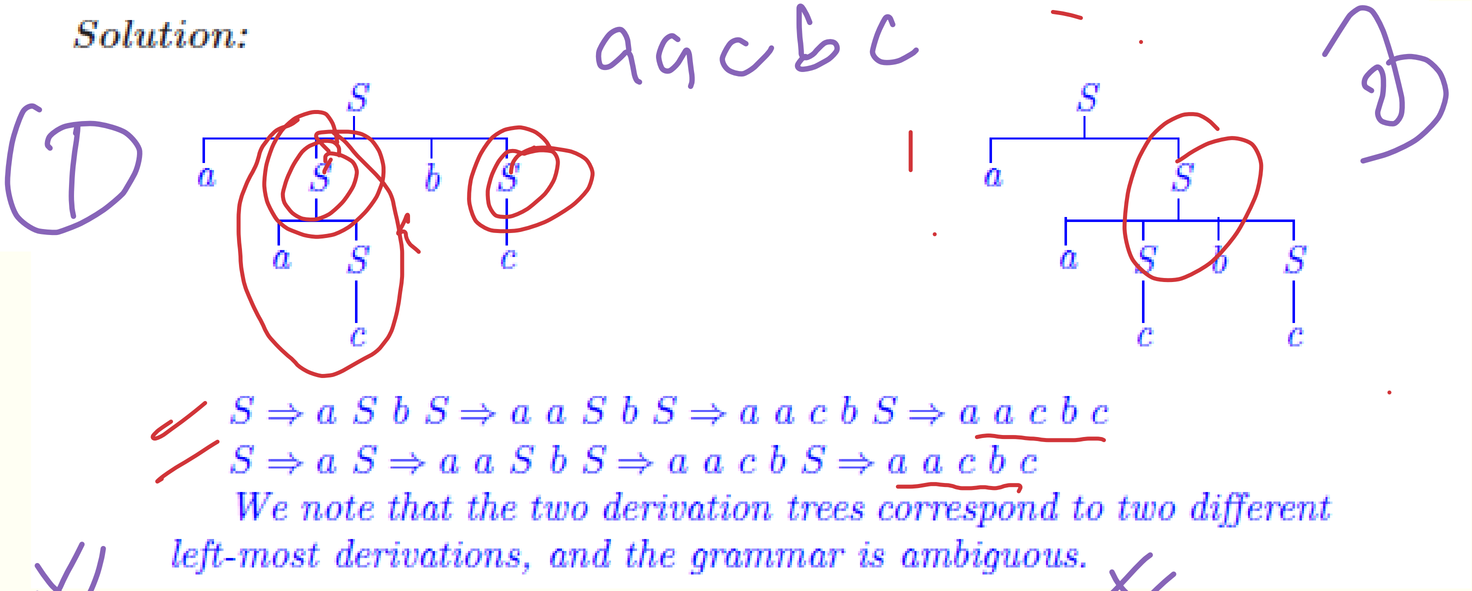
G2 ⇒ G1

Problem



Determine whether the following grammar is ambiguous. If so, show two different derivation trees for the same string of terminals, and show a left-most derivation corresponding to each tree.

1. $S \to aSbS$
2. $S \to aS$
3. $S \to c$



Solution:

aacbc

$S \Rightarrow a\ S\ b\ S \Rightarrow a\ a\ S\ b\ S \Rightarrow a\ a\ c\ b\ S \Rightarrow a\ a\ c\ b\ c$

$S \Rightarrow a\ S \Rightarrow a\ a\ S\ b\ S \Rightarrow a\ a\ c\ b\ S \Rightarrow a\ a\ c\ b\ c$

We note that the two derivation trees correspond to two different left-most derivations, and the grammar is ambiguous.

THE PARSING PROBLEM

Parsing Algorithm

- A parsing algorithm is needed in the syntax analysis phase of a compiler
- Given a grammar and a string of input symbols, determine whether the string belongs to the language of the grammar, and, if so, determine its structure.
- The two major classifications of parsing algorithms are top-down, and bottom-up, corresponding to the sequence in which a derivation tree is built or traversed

↓ ≠ ⇑

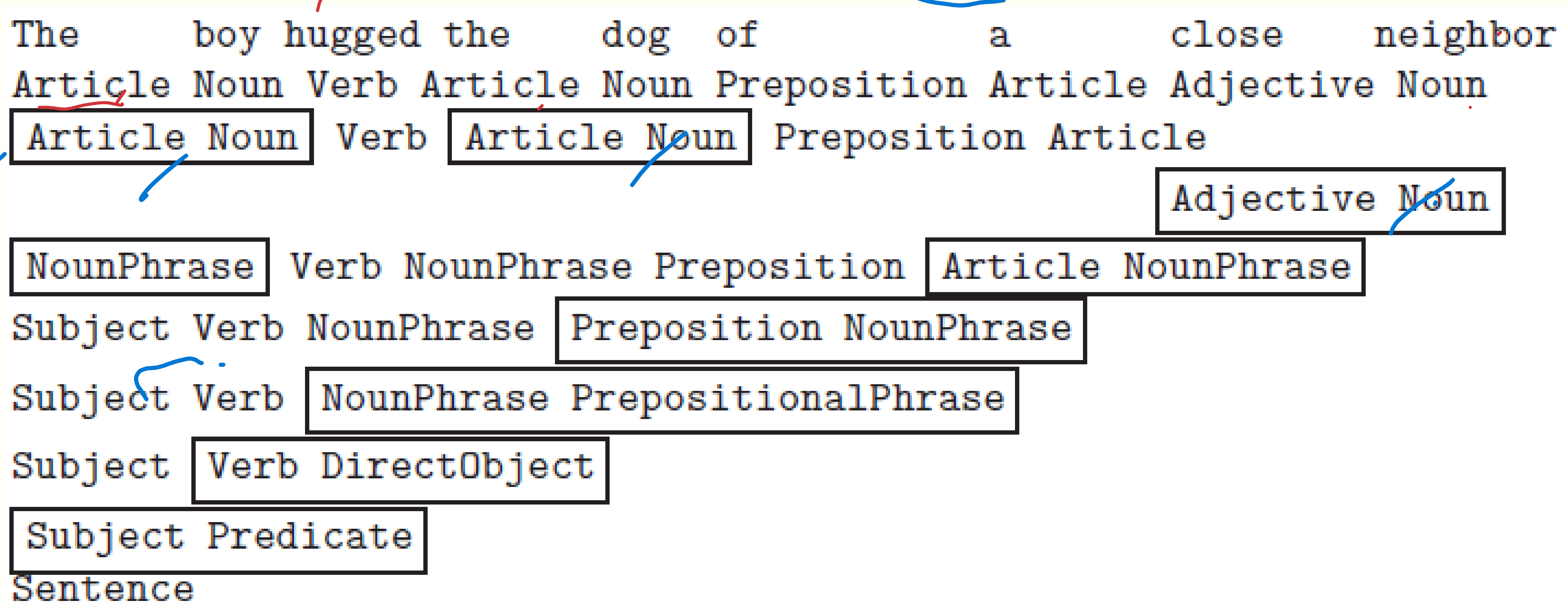
SableCC

⇑

Example: Parsing English Sentence

Given a grammar, G, and a string of input symbols, decide whether the string is in L(G); also, determine the structure of the input string.

The solution to the parsing problem will be 'yes' or 'no', and, if 'yes', some description of the input string's structure, such as a derivation tree.



The boy hugged the dog of a close neighbor

Article Noun Verb Article Noun Preposition Article Adjective Noun

Article Noun Verb Article Noun Preposition Article Adjective Noun

NounPhrase Verb NounPhrase Preposition Article NounPhrase

Subject Verb NounPhrase Preposition NounPhrase

Subject Verb NounPhrase PrepositionalPhrase

Subject Verb DirectObject

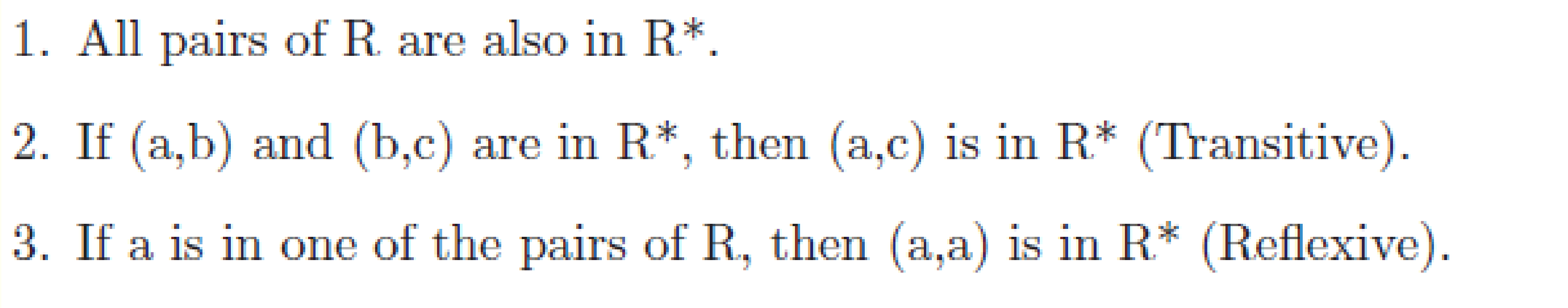
Subject Predicate

Sentence

RELATION AND CLOSURE

Definition

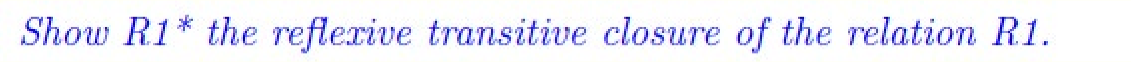
- The closure of a relation R with respect to property P is the relation obtained by adding the minimum number of ordered pairs to R to obtain property P.
- If R is a relation, then the reflexive transitive closure of R is designated R*;
- 3 types of closure: reflexive, symmetric and transitive.



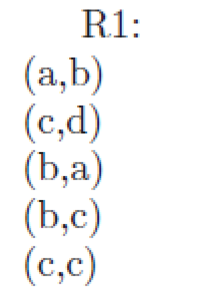
1. All pairs of R are also in R*.
2. If (a,b) and (b,c) are in R*, then (a,c) is in R* (Transitive).
3. If a is in one of the pairs of R, then (a,a) is in R* (Reflexive).

- So, reflexive is adding loop and transitive adding a new arc

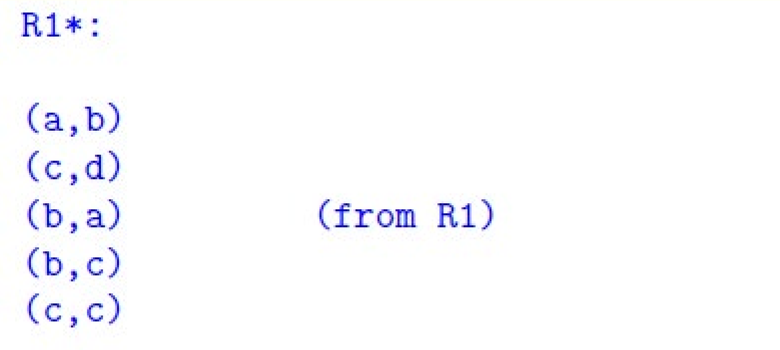
Example



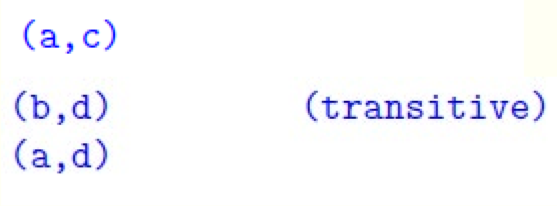
Show R1 the reflexive transitive closure of the relation R1.*



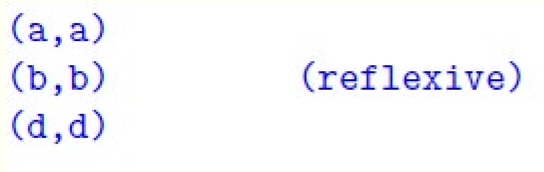
R1:
(a,b)
(c,d)
(b,a)
(b,c)
(c,c)



```
R1*:

(a,b)
(c,d)
(b,a)           (from R1)
(b,c)
(c,c)
```



```
(a,c)

(b,d)           (transitive)
(a,d)
```



```
(a,a)
(b,b)           (reflexive)
(d,d)
```

SIMPLE GRAMMARS

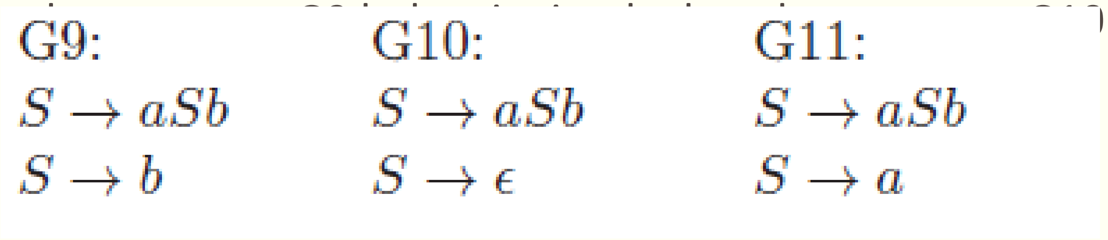
Definition

A grammar is a simple grammar if every rule is of the form:

$$A \rightarrow a\alpha$$

- where A represents any nonterminal, a represents any terminal, and α represents any string of terminals and nonterminals
- and every pair of rules defining the same nonterminal begin with different terminals on the right side of the arrow.

Example

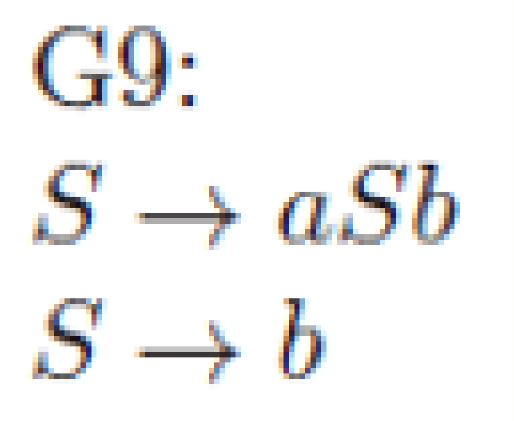


G9:	G10:	G11:
$S \rightarrow aSb$	$S \rightarrow aSb$	$S \rightarrow aSb$
$S \rightarrow b$	$S \rightarrow \epsilon$	$S \rightarrow a$

- the grammar G9 below is simple, but the grammar G10 is not simple because it contains an epsilon rule
- the grammar G11 is not simple because two rules defining S begin with the same terminal.

Selection set

- The set of input symbols (i.e. terminal symbols) which imply the application of a grammar rule
- For simple grammars, the selection set of each rule always contains exactly one terminal symbol - the one beginning the right-hand side.
- In grammar G9, the selection set of the first rule is {a} and the selection set of the second rule is {b}.

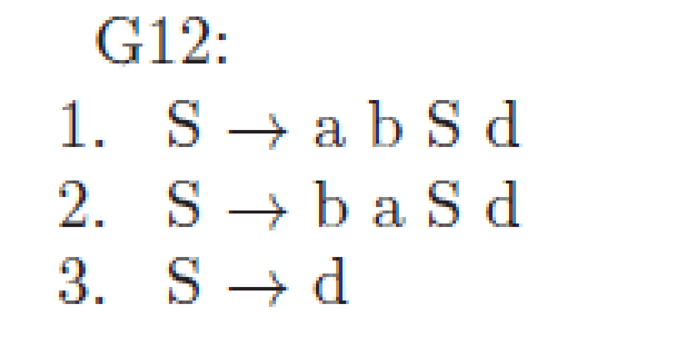


G9:

$$S \rightarrow aSb$$

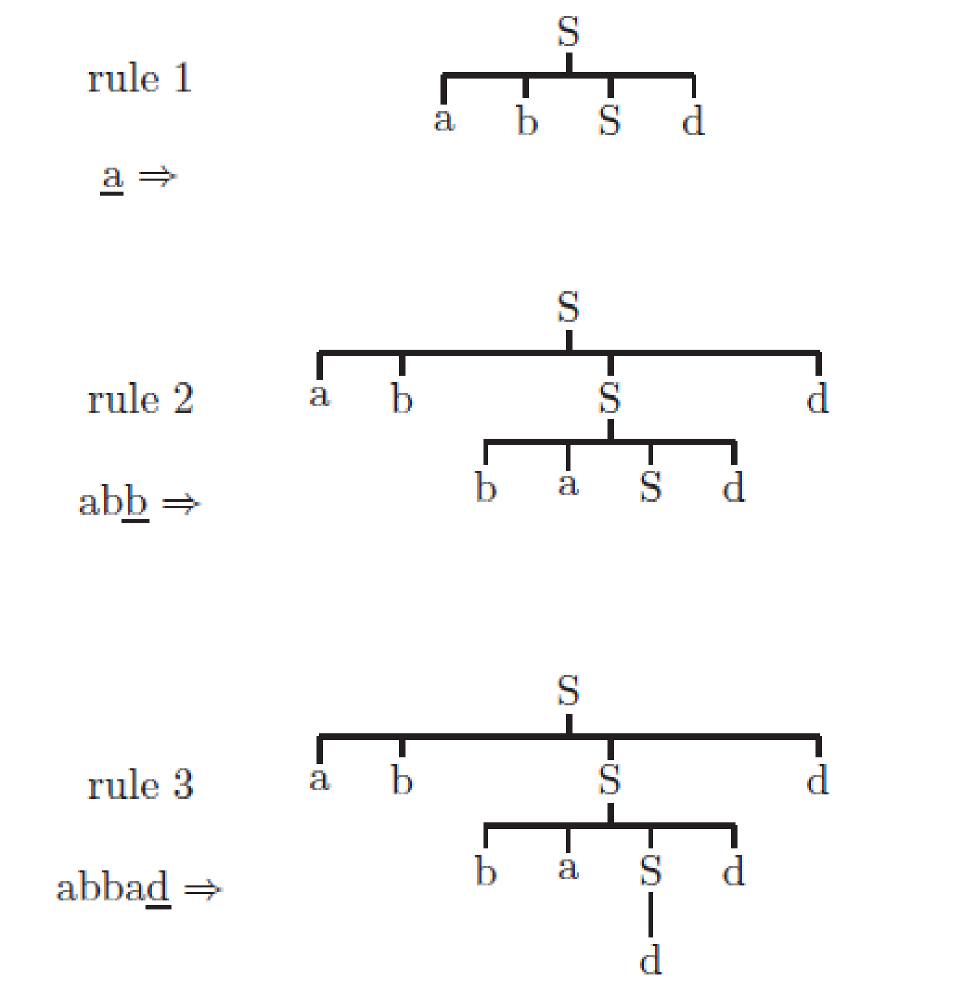
$$S \rightarrow b$$

Example 2: Derivation tree for abbaddd

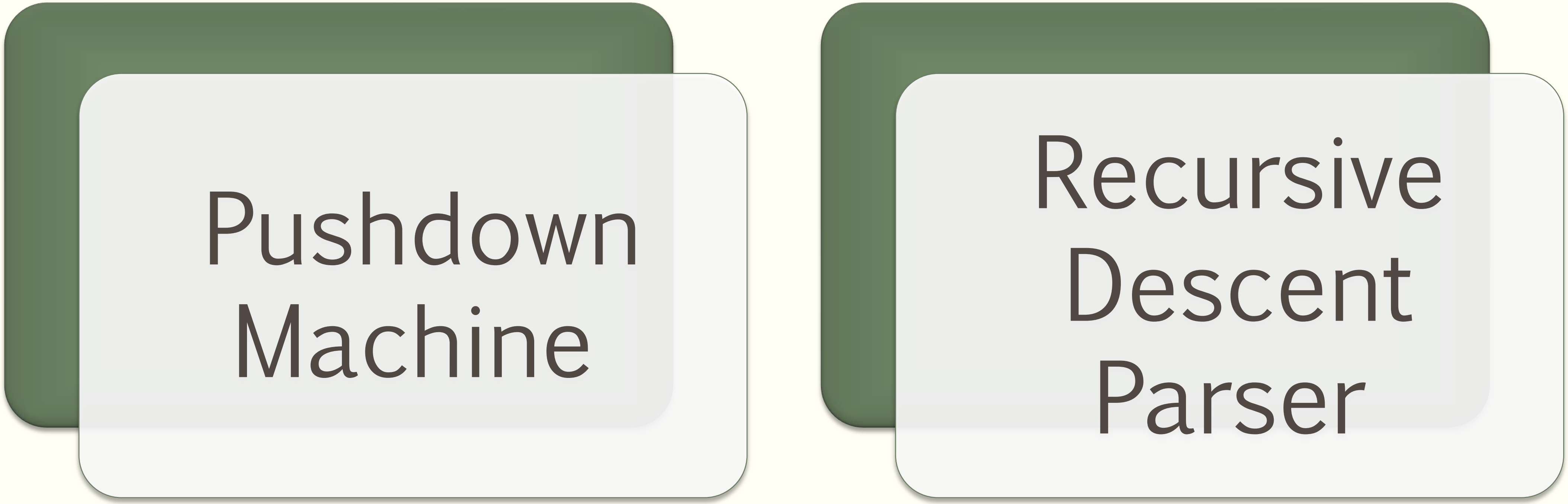


G12:

1. S → a b S d
2. S → b a S d
3. S → d



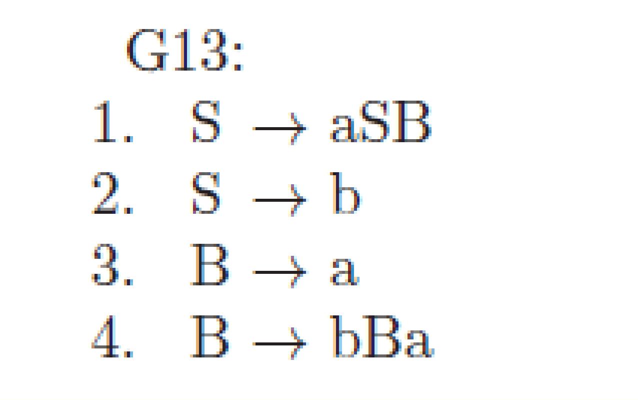
Parsing simple language with Pushdown Machines



Pushdown Machine

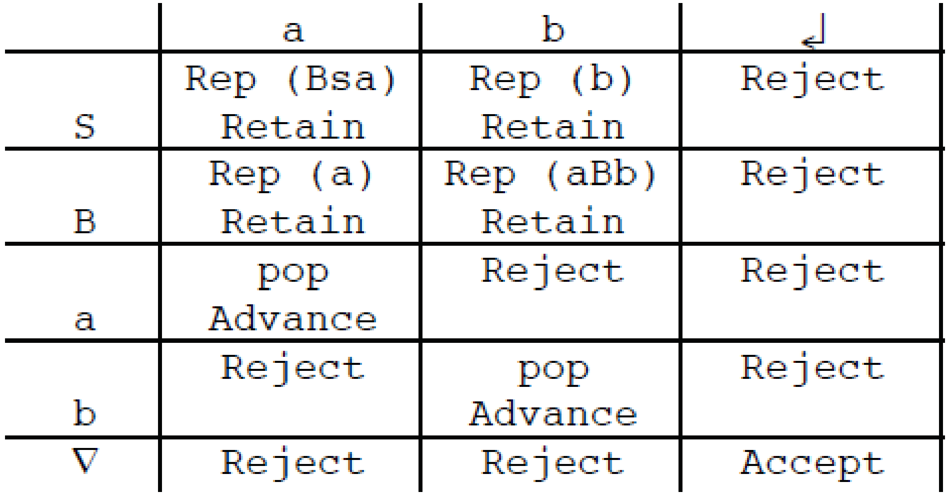
Recursive Descent Parser

Parsing simple language with Pushdown Machines

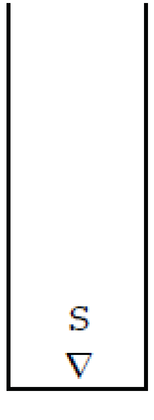


G13:

1. S → aSB
2. S → b
3. B → a
4. B → bBa

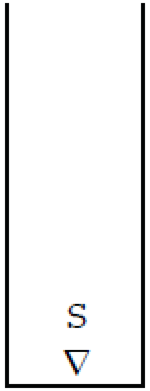


	a	b	↵
S	Rep (Bsa) Retain	Rep (b) Retain	Reject
B	Rep (a) Retain	Rep (aBb) Retain	Reject
a	pop Advance	Reject	Reject
b	Reject	pop Advance	Reject
∇	Reject	Reject	Accept

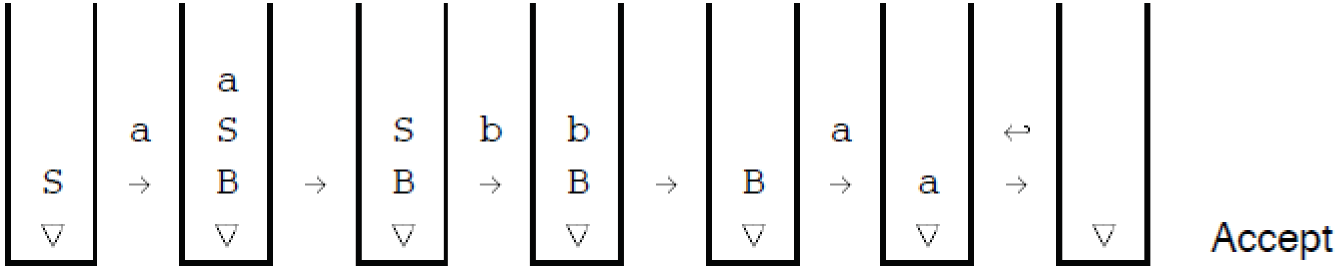


Initial

	a	b	↲
S	Rep (Bsa) Retain	Rep (b) Retain	Reject
B	Rep (a) Retain	Rep (aBb) Retain	Reject
a	pop Advance	Reject	Reject
b	Reject	pop Advance	Reject
∇	Reject	Reject	Accept



Initial



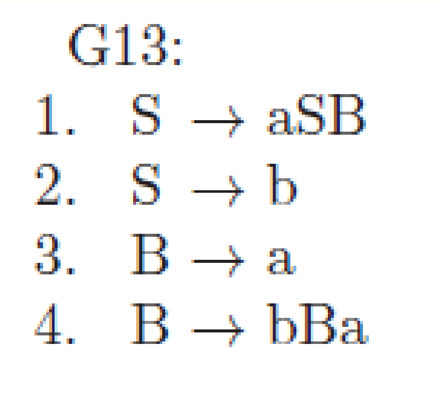
Recursive Descent Parser for Simple Grammar

the parser is written using a traditional programming language, such as Java or C++.

A method is written for each nonterminal in the grammar.

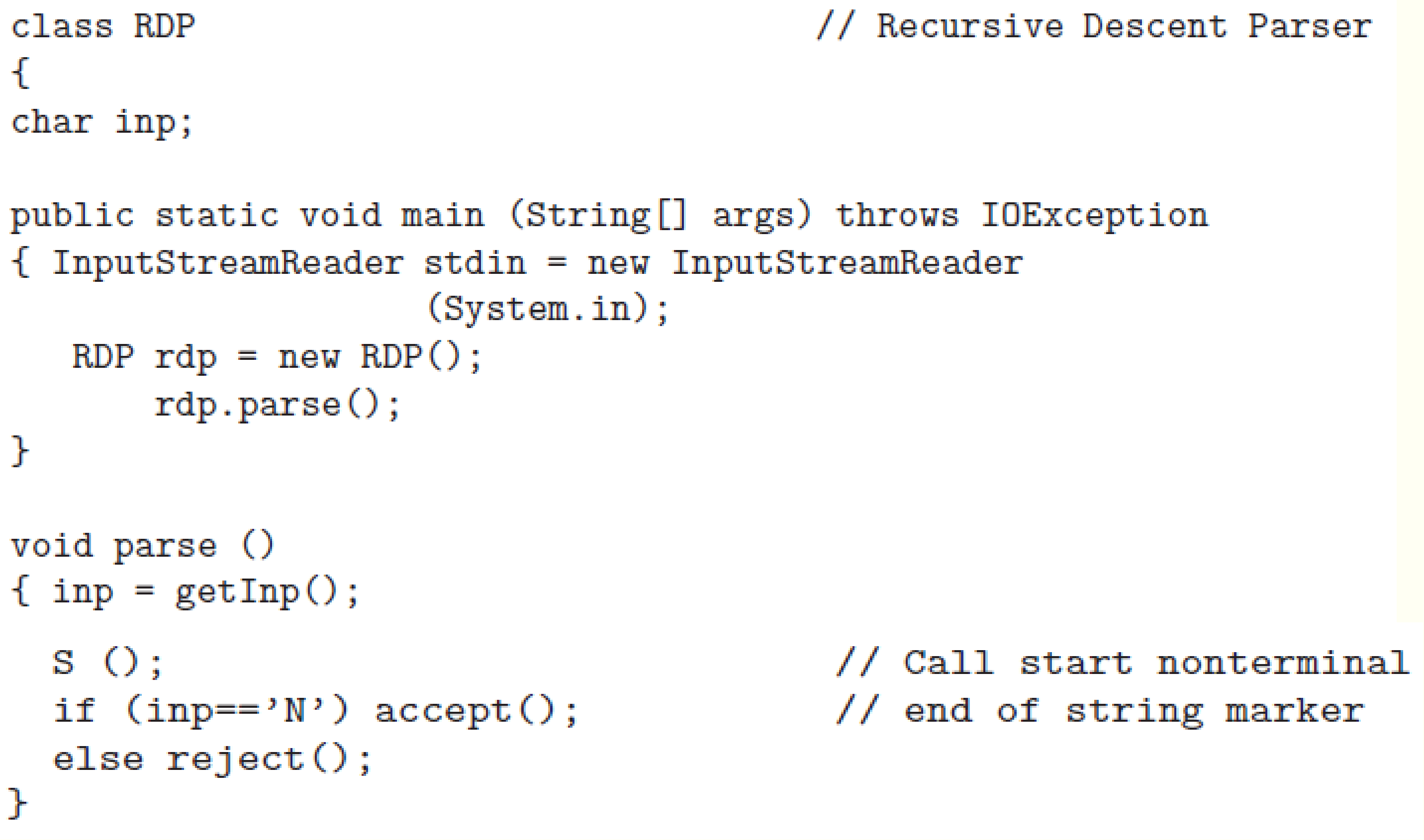
it handles nonterminals by calling the corresponding methods, and it handles terminals by reading another input symbol.

Example



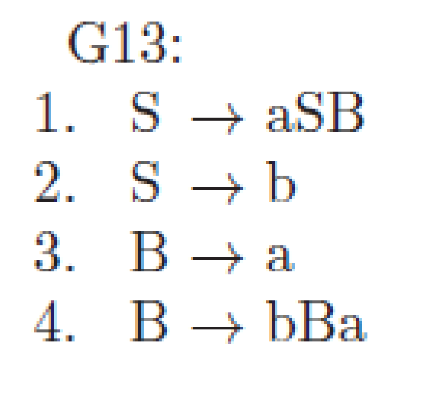
G13:

1. $S \rightarrow aSB$
2. $S \rightarrow b$
3. $B \rightarrow a$
4. $B \rightarrow bBa$



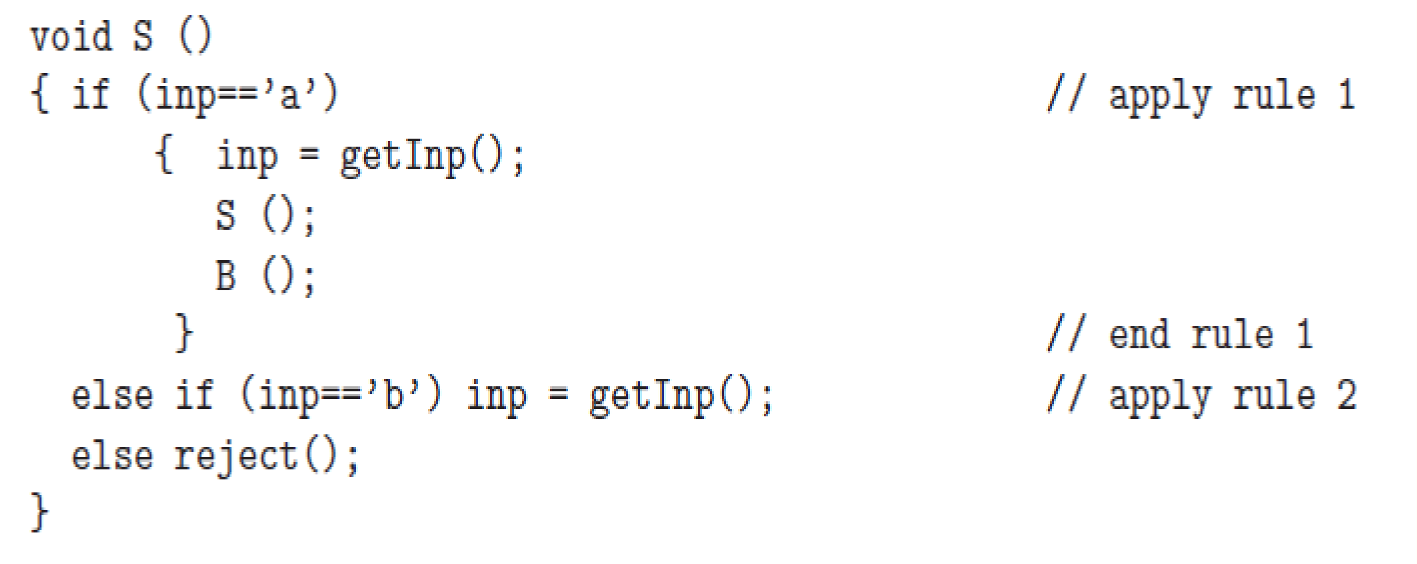
```
class RDP                                      // Recursive Descent Parser
{
char inp;

public static void main (String[] args) throws IOException
{ InputStreamReader stdin = new InputStreamReader
                    (System.in);
   RDP rdp = new RDP();
       rdp.parse();
}

void parse ()
{ inp = getInp();
  S ();                                         // Call start nonterminal
  if (inp=='N') accept();                       // end of string marker
  else reject();
}
```

Continue



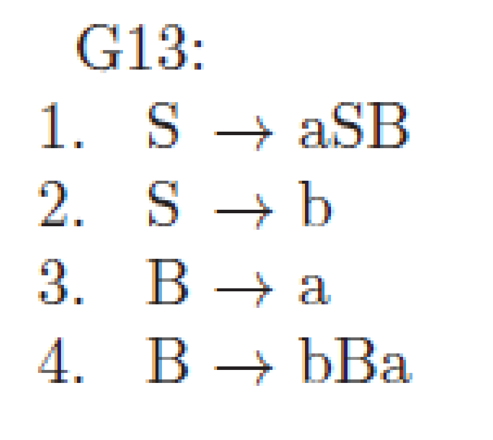
G13:

1. S → aSB
2. S → b
3. B → a
4. B → bBa



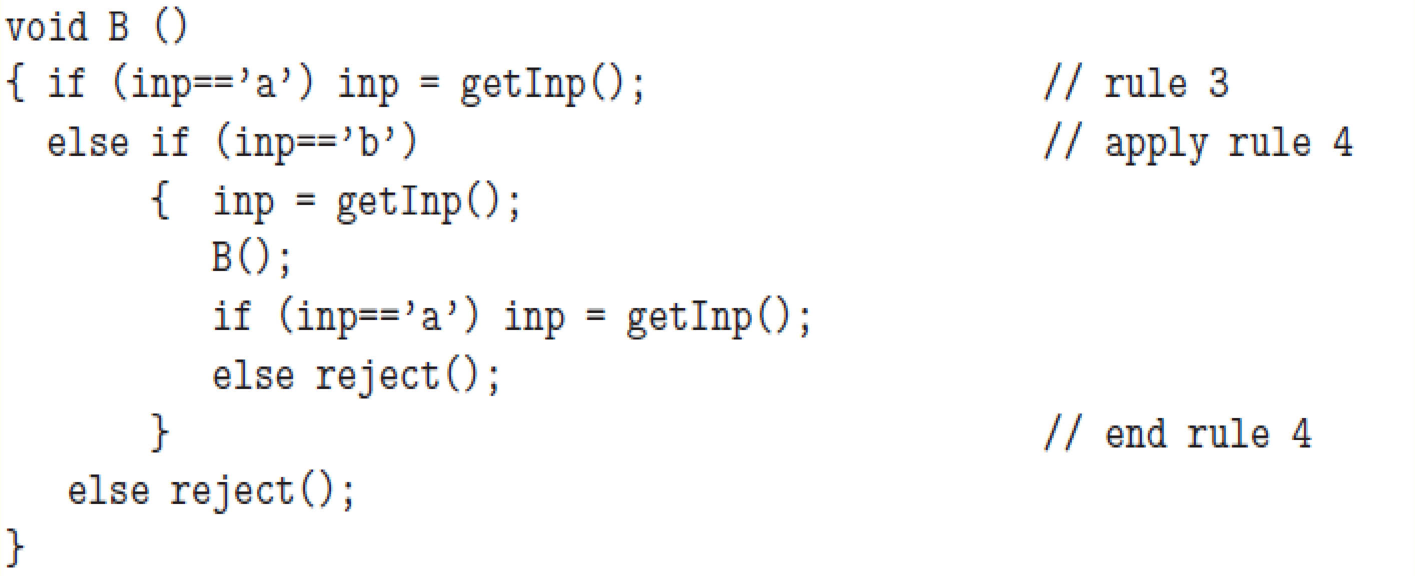
```
void S ()
{ if (inp==’a’)                                   // apply rule 1
      {  inp = getInp();
         S ();
         B ();
       }                                          // end rule 1
  else if (inp==’b’) inp = getInp();              // apply rule 2
  else reject();
}
```

Continue



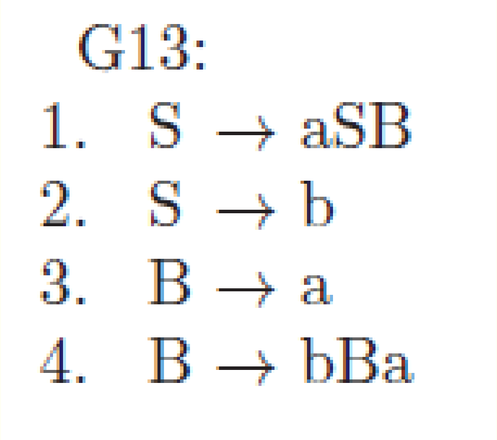
G13:

1. $S \rightarrow aSB$
2. $S \rightarrow b$
3. $B \rightarrow a$
4. $B \rightarrow bBa$



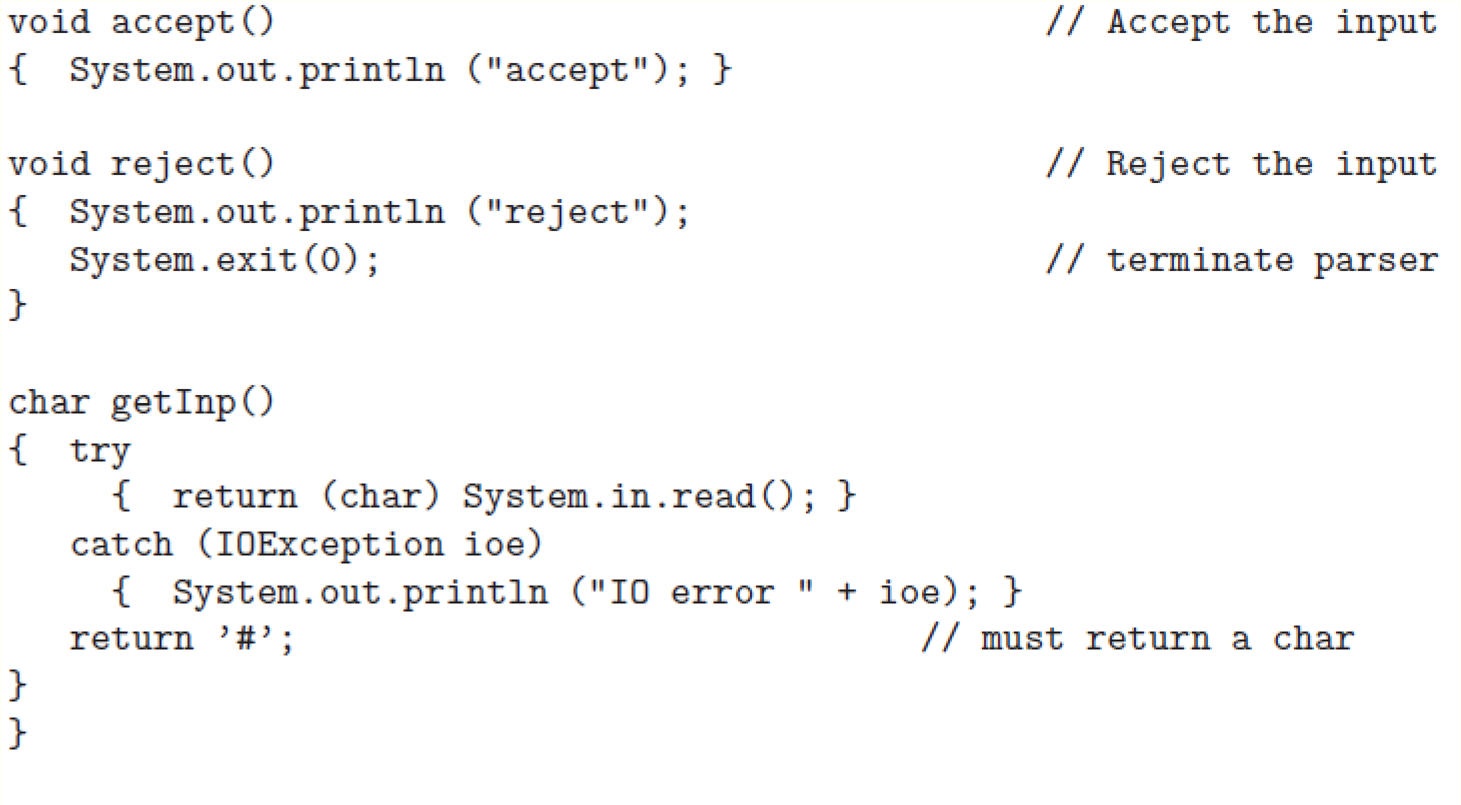
```
void B ()
{ if (inp==’a’) inp = getInp();                   // rule 3
  else if (inp==’b’)                              // apply rule 4
       {  inp = getInp();
          B();
          if (inp==’a’) inp = getInp();
          else reject();
       }                                          // end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

Continue



G13:

1. $S \rightarrow aSB$
2. $S \rightarrow b$
3. $B \rightarrow a$
4. $B \rightarrow bBa$



```
void accept()                                     // Accept the input
{  System.out.println ("accept"); }

void reject()                                     // Reject the input
{  System.out.println ("reject");
   System.exit(0);                                // terminate parser
}

char getInp()
{  try
     {  return (char) System.in.read(); }
   catch (IOException ioe)
     {  System.out.println ("IO error " + ioe); }
   return ’#’;                              // must return a char
}
}
```

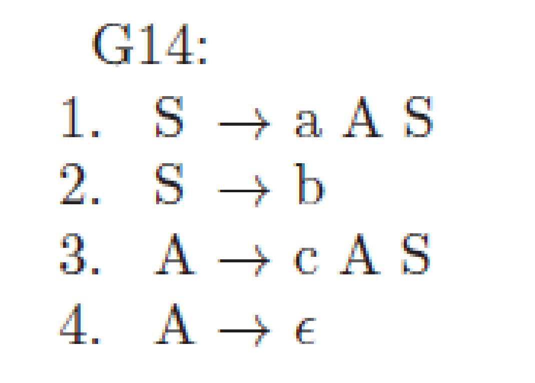
QUASI SIMPLE GRAMMARS

Definition

A quasi-simple grammar is a grammar which obeys the restriction of simple grammars, but which may also contain rules of the form

$$N \rightarrow \epsilon$$

- where N represents any nonterminal as long as all rules defining the same nonterminal have disjoint selection sets.

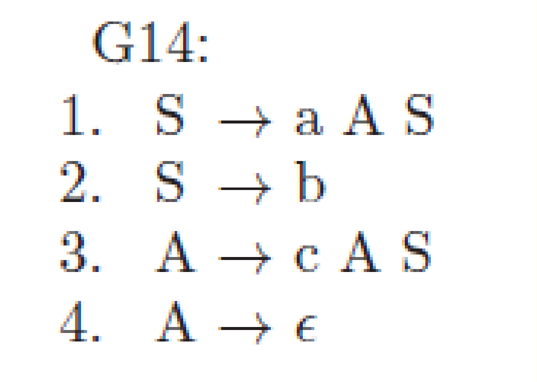


G14:

1. $S \rightarrow a\ A\ S$
2. $S \rightarrow b$
3. $A \rightarrow c\ A\ S$
4. $A \rightarrow \epsilon$

Follow set and Selection set

- The follow set of a nonterminal A, designated Fol(A), is the set of all terminals (or endmarker ← (which can immediately follow an A in an intermediate form derived from S ← ,where S is the starting nonterminal.

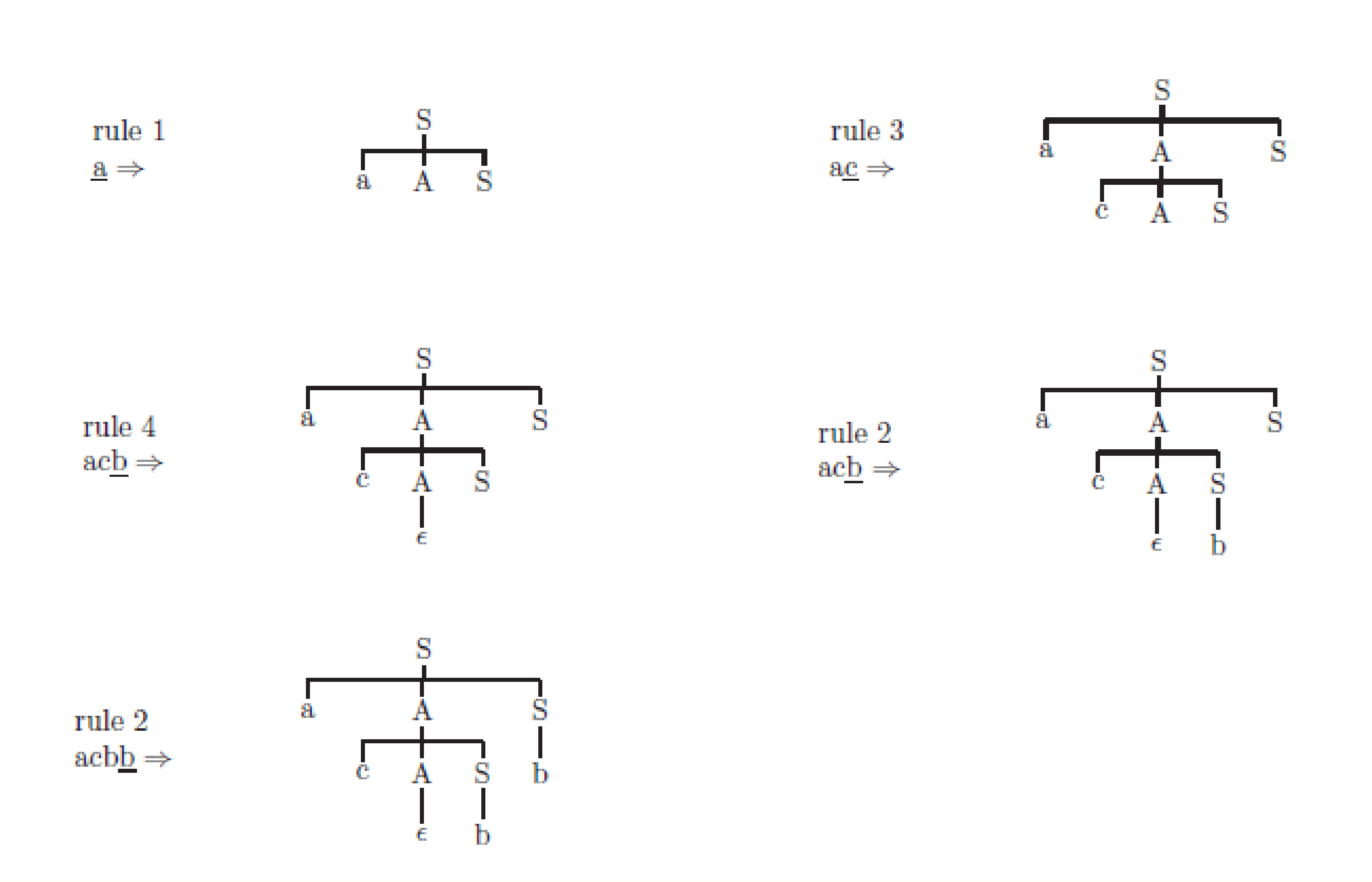


G14:

1. $S \rightarrow a\ A\ S$
2. $S \rightarrow b$
3. $A \rightarrow c\ A\ S$
4. $A \rightarrow \epsilon$

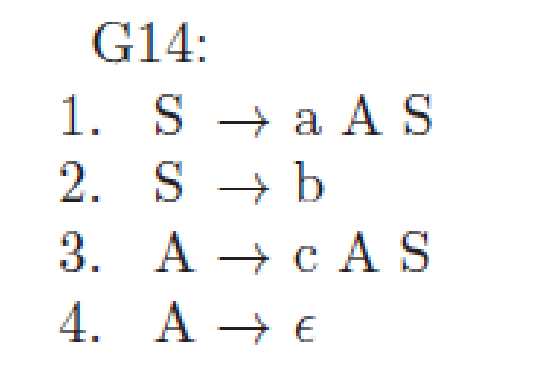
- the follow set of S is {a,b ↩} and the follow set of A is {a,b}

Build a derivation tree for the input string **acbb**



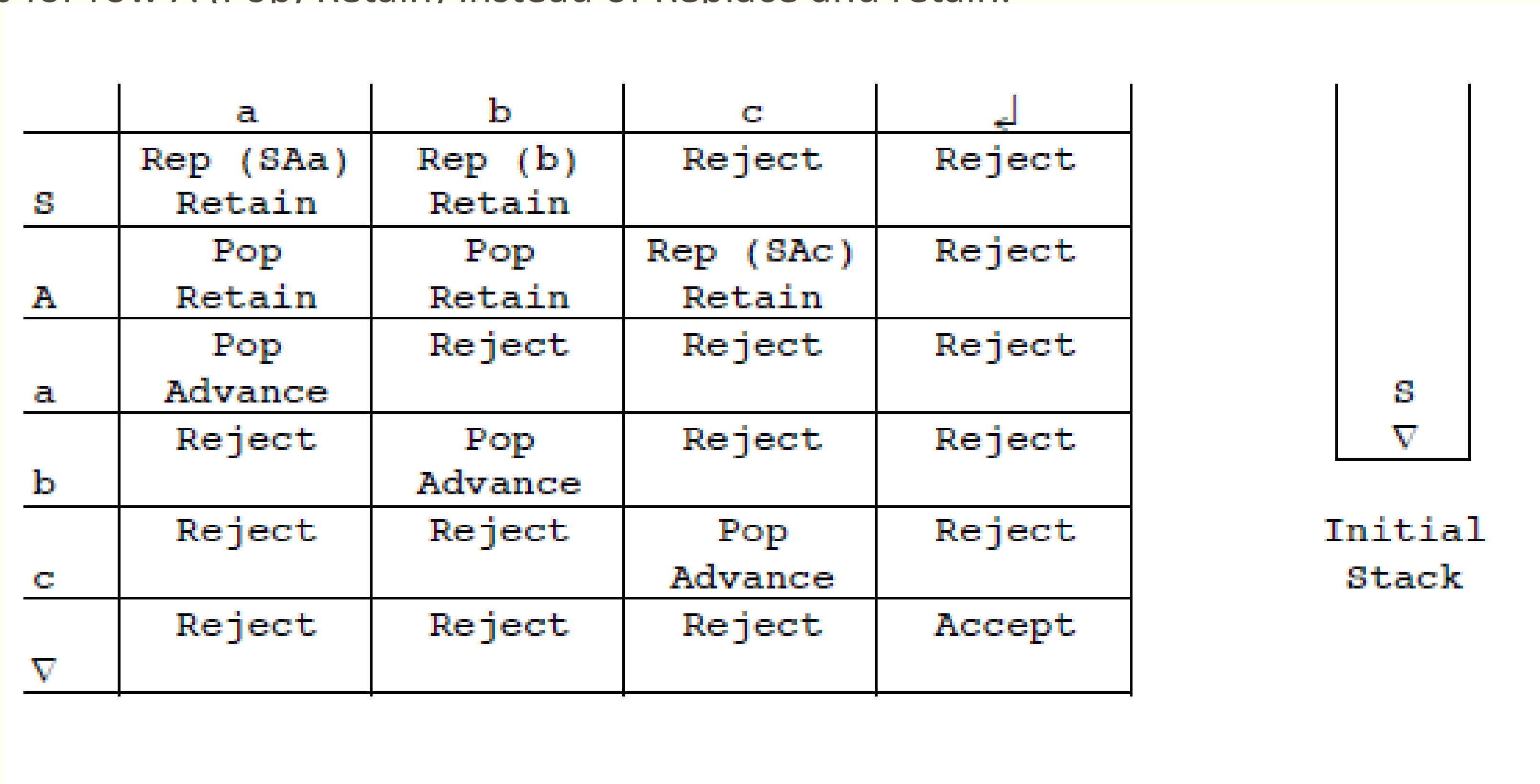
Pushdown Machine for Quasi-Simple Grammars

- The entries in columns a and b for row A (Pop, Retain) instead of Replace and retain.



G14:

1. $S \rightarrow a\ A\ S$
2. $S \rightarrow b$
3. $A \rightarrow c\ A\ S$
4. $A \rightarrow \epsilon$

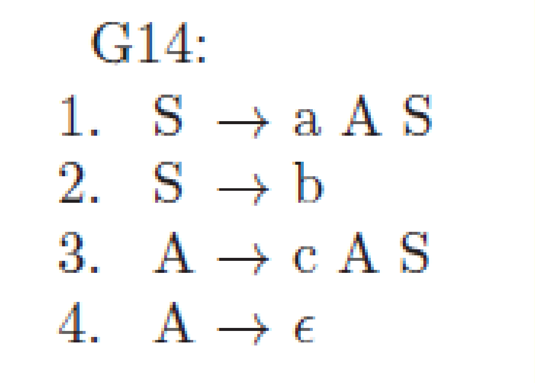


	a	b	c	↲
S	Rep (SAa) Retain	Rep (b) Retain	Reject	Reject
A	Pop Retain	Pop Retain	Rep (SAc) Retain	Reject
a	Pop Advance	Reject	Reject	Reject
b	Reject	Pop Advance	Reject	Reject
c	Reject	Reject	Pop Advance	Reject
∇	Reject	Reject	Reject	Accept

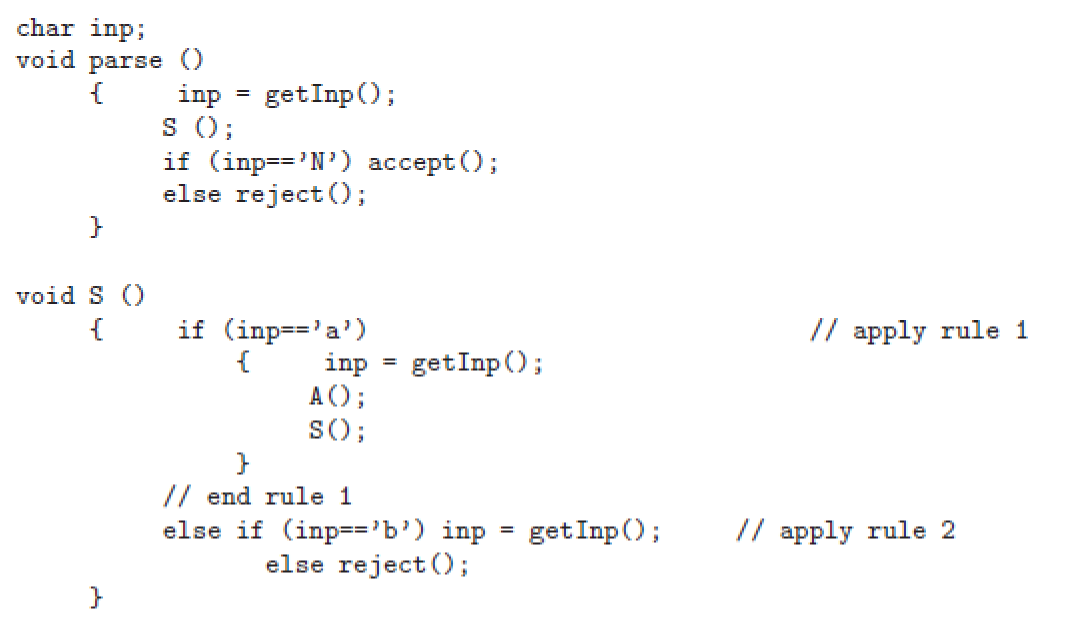
S
∇

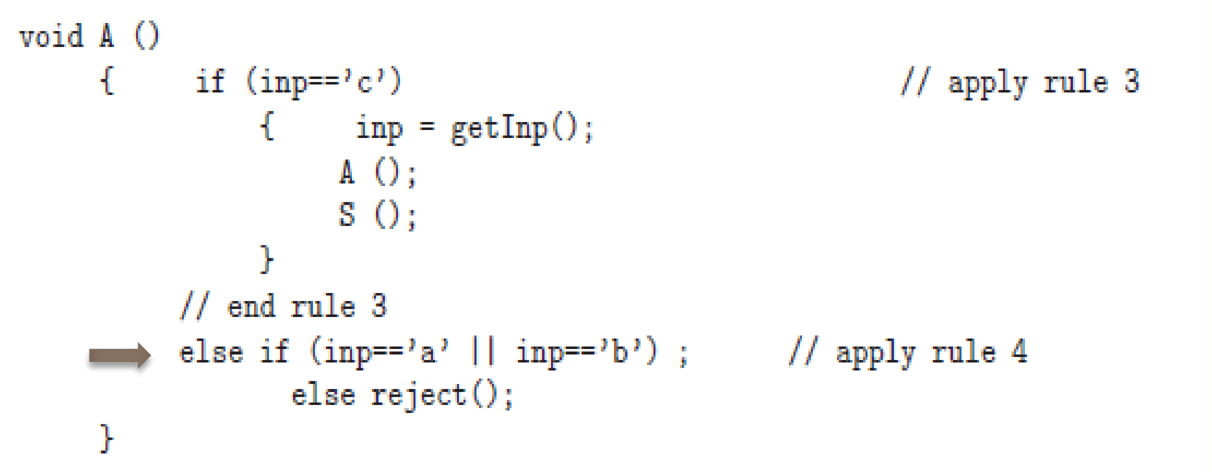
Initial
Stack

Recursive Descent for Quasi Simple Grammars


G14:

1. $S \rightarrow a\ A\ S$
2. $S \rightarrow b$
3. $A \rightarrow c\ A\ S$
4. $A \rightarrow \epsilon$


```
char inp;
void parse ()
     {     inp = getInp();
          S ();
          if (inp=='N') accept();
          else reject();
     }

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


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

LL(1) GRAMMARS

Recap

- Simple grammars: $\underline{A} \to \underline{a}\alpha$

- Quasi-simple grammars: $A \to a\alpha$
  $\mathrm{N} \to \epsilon$

- LL(1) Grammars $\underline{N} \to \underline{\alpha}$

- the grammar must be such that any two rules defining the same nonterminal must have disjoint selection sets,

- we can construct a one-state pushdown machine parser or a recursive descent parser for it.

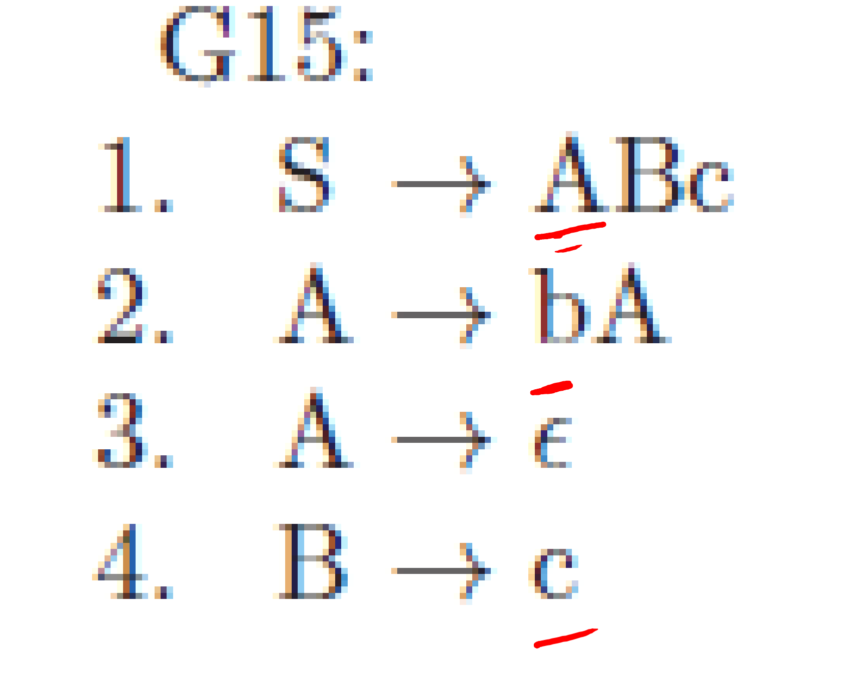
Definition

- A context-free grammar $G = (V_T, V_N, S, P)$ whose parsing table has no multiple entries is said to be $LL(1)$.

- In the name $LL(1)$
  - the first L stands for scanning the input from **l**eft to right,
  - the second L stands for producing a **l**eftmost derivation,
  - and the 1 stands for using **one** input symbol of lookahead at each step to make parsing action decision.

Find Selection set

Nl → φ



G15:

1. $S \rightarrow \underline{A}Bc$
2. $A \rightarrow bA$
3. $A \rightarrow \epsilon$
4. $B \rightarrow \underline{c}$

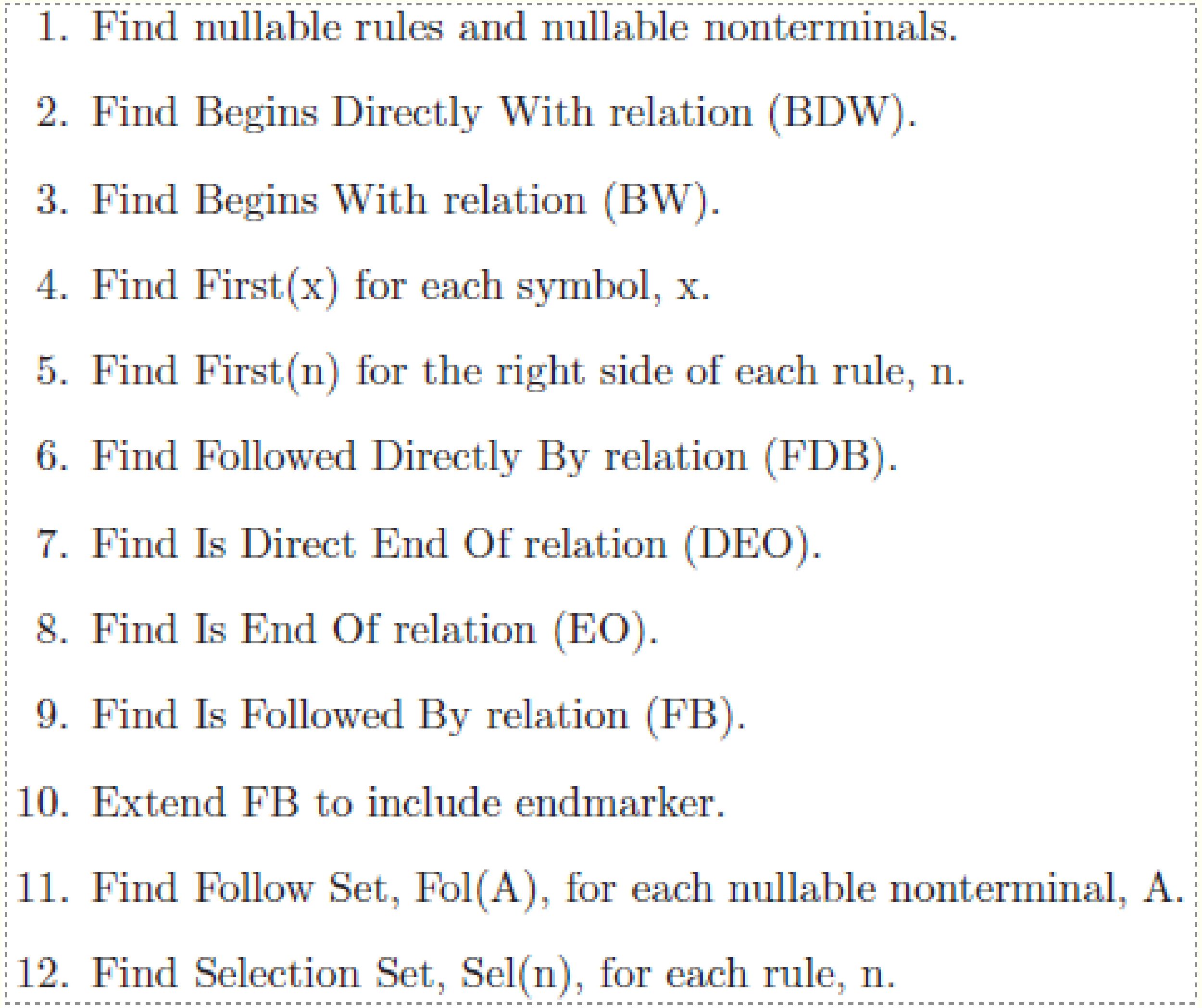
?

{b} ✓

= ? fol (A)

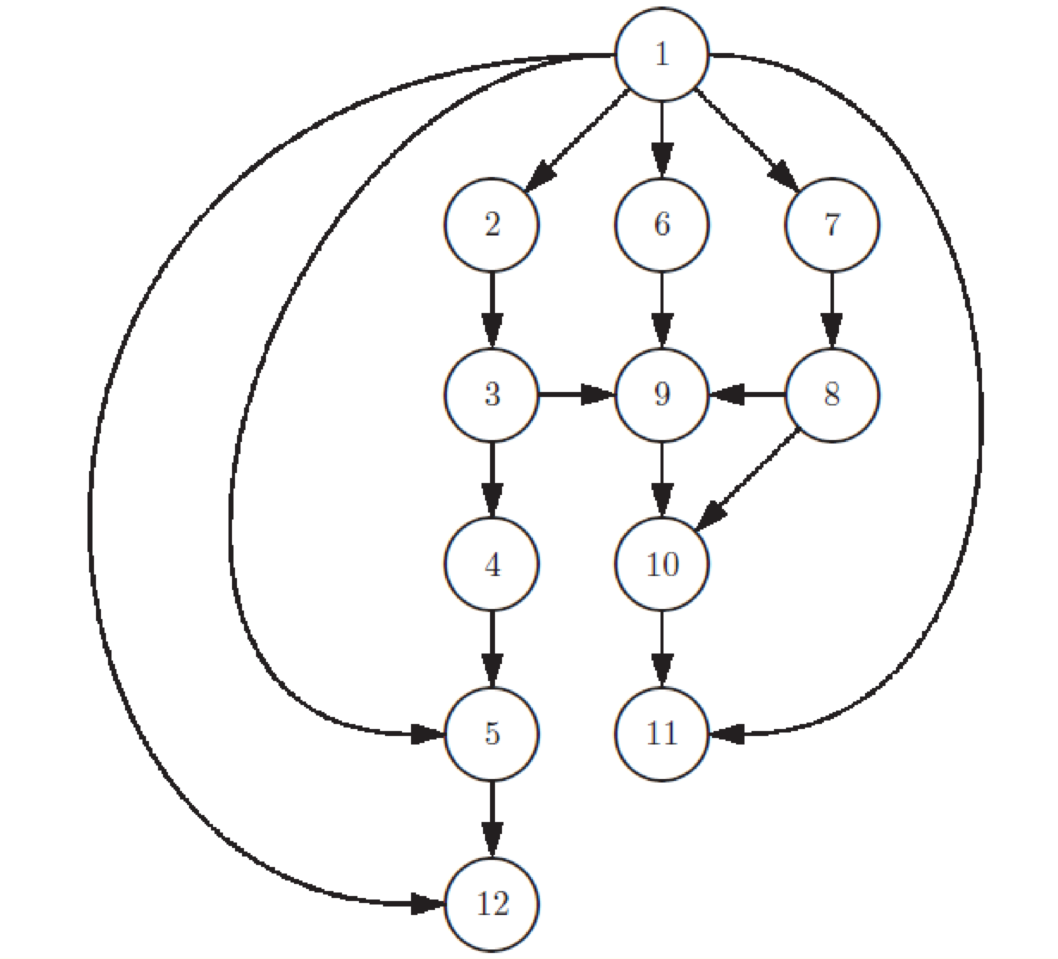
{c} ✓

12 steps to find selection set for LL(1) Grammars



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.

Dependency graph for the steps in the algorithm for finding selection sets



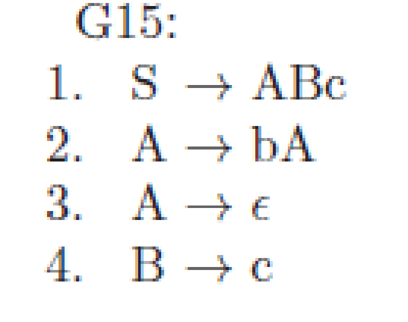
STEP 1: Find all nullable rules and nullable nonterminals:

- All Ɛ rules are nullable rules.
- The nonterminal defined in a nullable rule is a nullable nonterminal
- all rules in the form

$$A \rightarrow BCD...$$

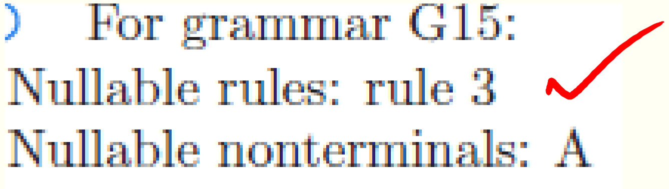
where B, C, D, … are all nullable non-terminals, are nullable rules; the non-terminals defined by these rules are also nullable non-terminals.

In other words, a nonterminal is nullable if Ɛ can be derived from it, and a rule is nullable if Ɛ can be derived from its right side.



G15:

1. $S \rightarrow ABc$
2. $A \rightarrow bA$
3. $A \rightarrow \epsilon$
4. $B \rightarrow c$

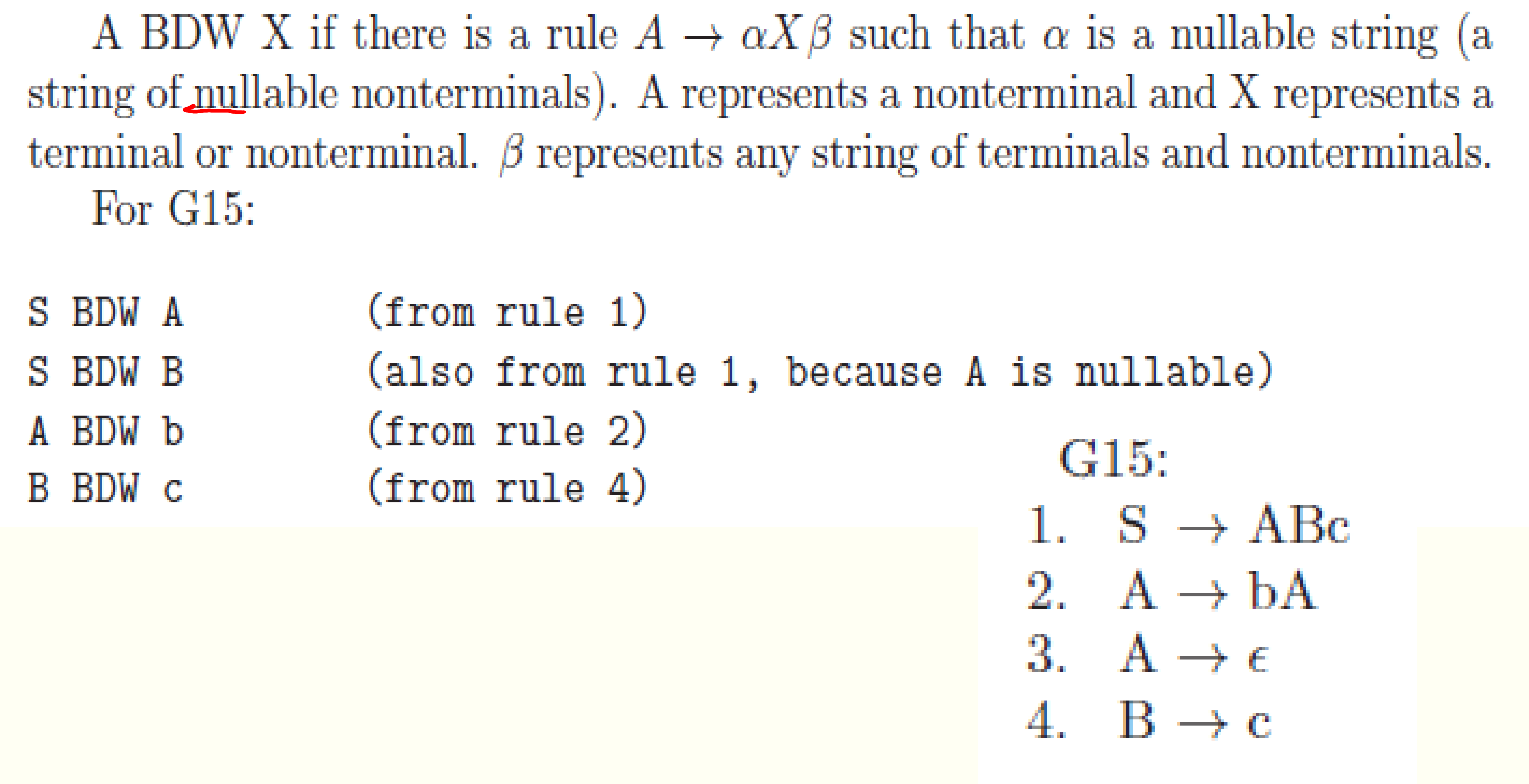


For grammar G15:

Nullable rules: rule 3 ✓

Nullable nonterminals: A

Step 2: Compute the relation Begins Directly With for each non-terminal



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.

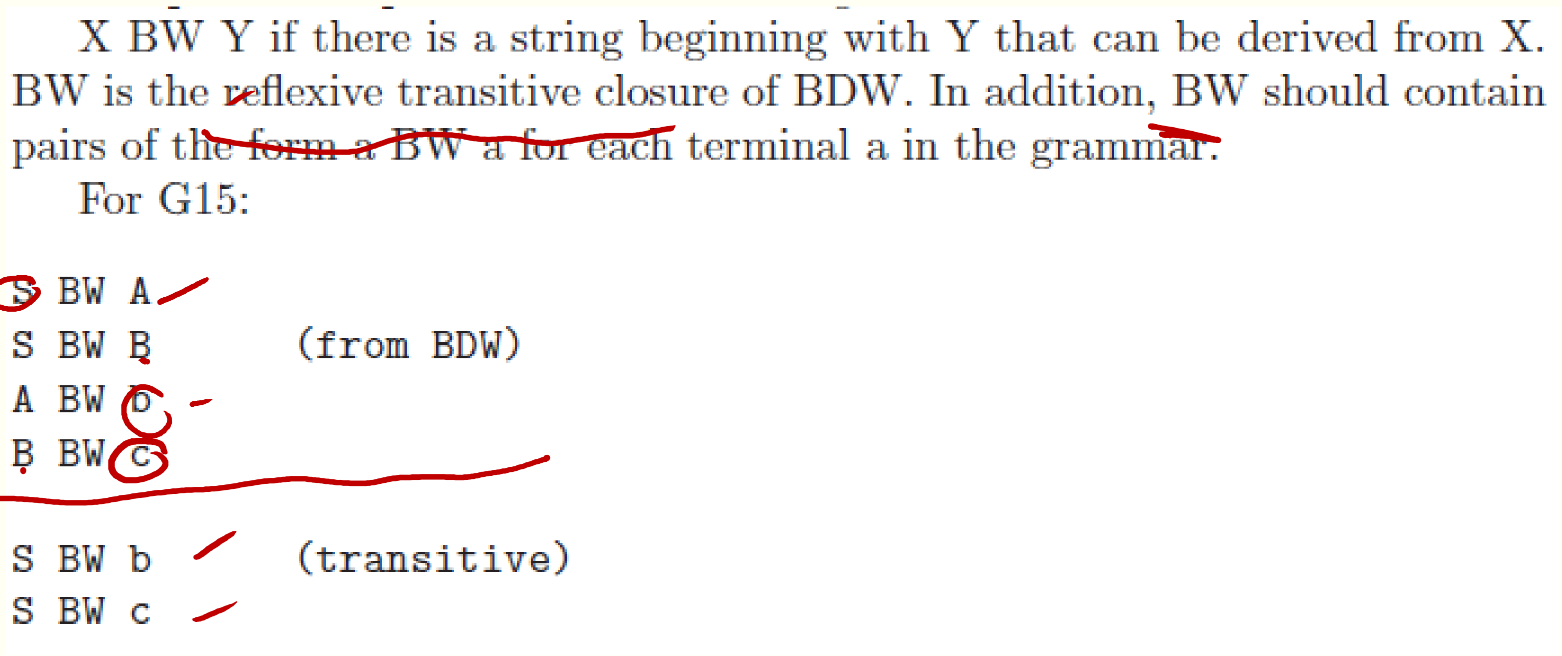
For G15:

```
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)
```

G15:

1. $S \rightarrow ABc$
2. $A \rightarrow bA$
3. $A \rightarrow \epsilon$
4. $B \rightarrow c$

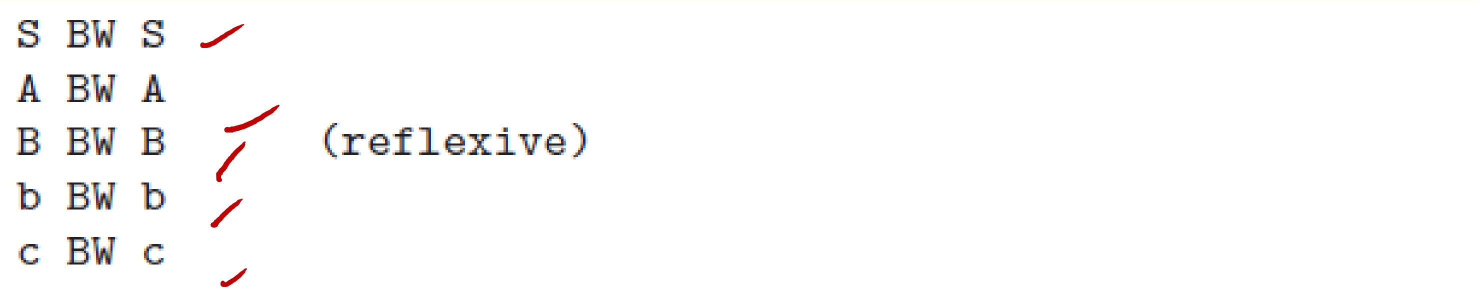
Step 3: Compute the relation Begins With



X BW Y if there is a string beginning with Y that can be derived from X. 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.

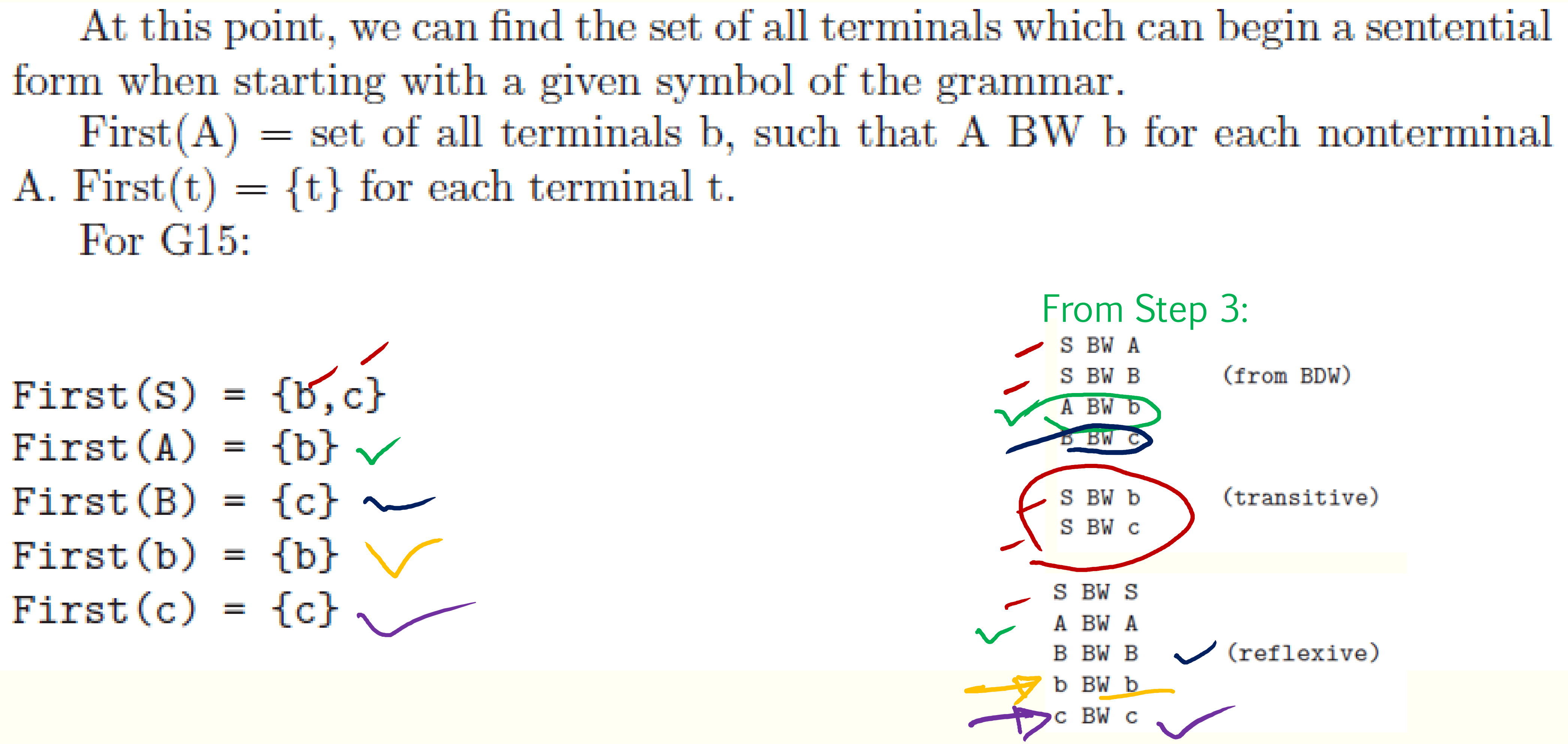
For G15:

```
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 the set of terminals First(x) for each symbol x in the grammar



At this point, we can find the set of all terminals which can begin a sentential form when starting with a given symbol of 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.

For G15:

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

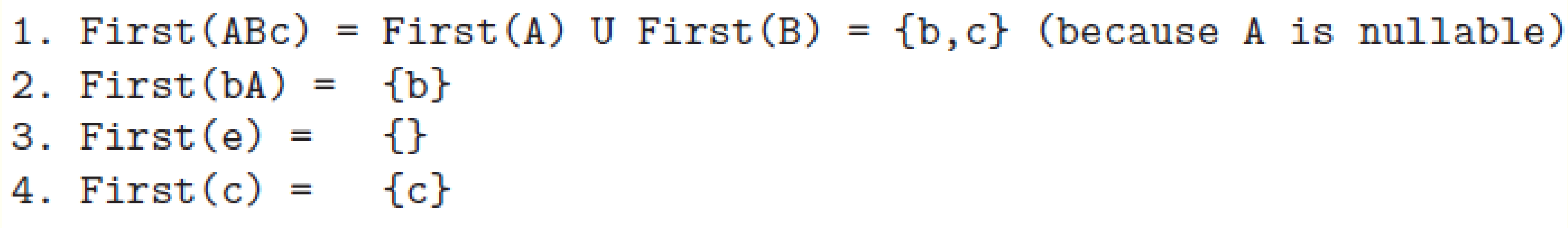
From Step 3:

```
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 5: Compute First of right side of each rule

- We now compute the set of terminals which can begin a sentential form derivable from the right side of each rule.

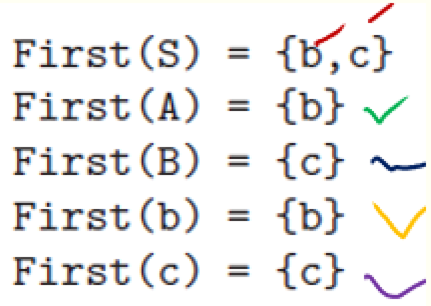
- In other words, find the union of the First(x) sets for each symbol on the right side of a rule, but stop when reaching a non-nullable symbol.



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

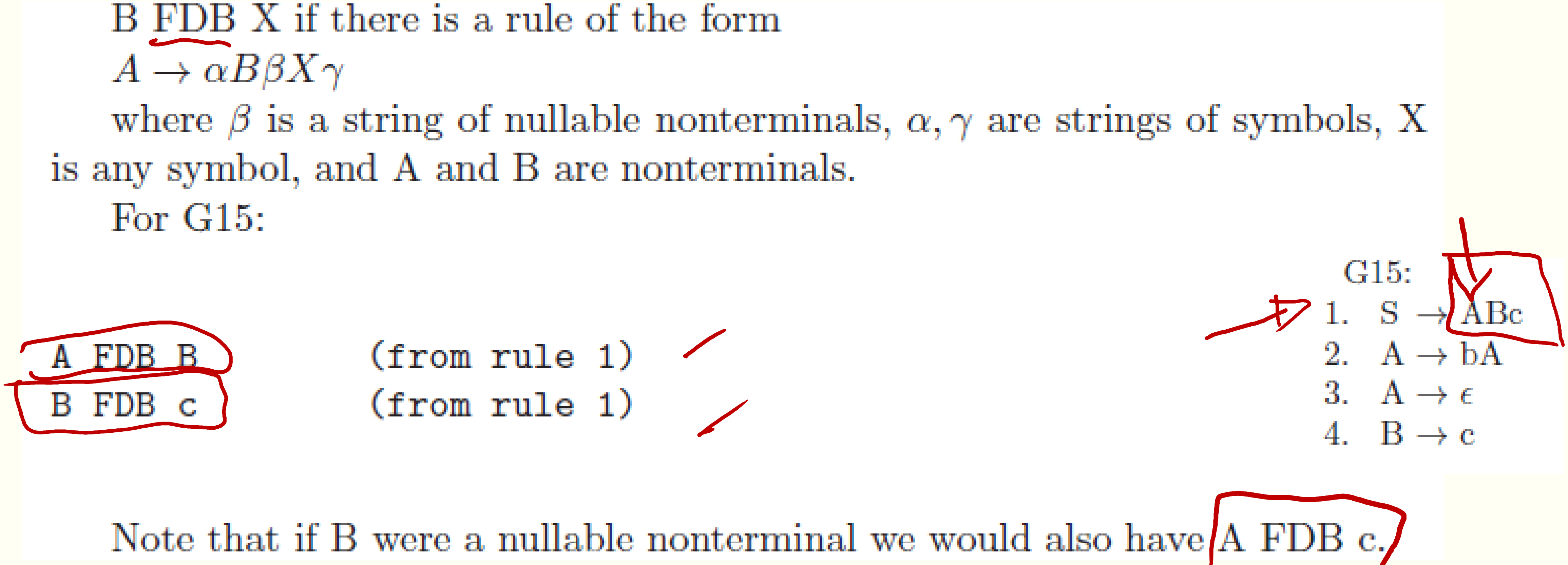
- If the grammar contains no nullable rules, you may skip to step 12 at this point.

S4 →



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

Step 6: Compute the relation Is Followed Directly By



B FDB X if there is a rule of the form

$A \to \alpha B \beta X \gamma$

where β is a string of nullable nonterminals, α, γ are strings of symbols, X is any symbol, and A and B are nonterminals.

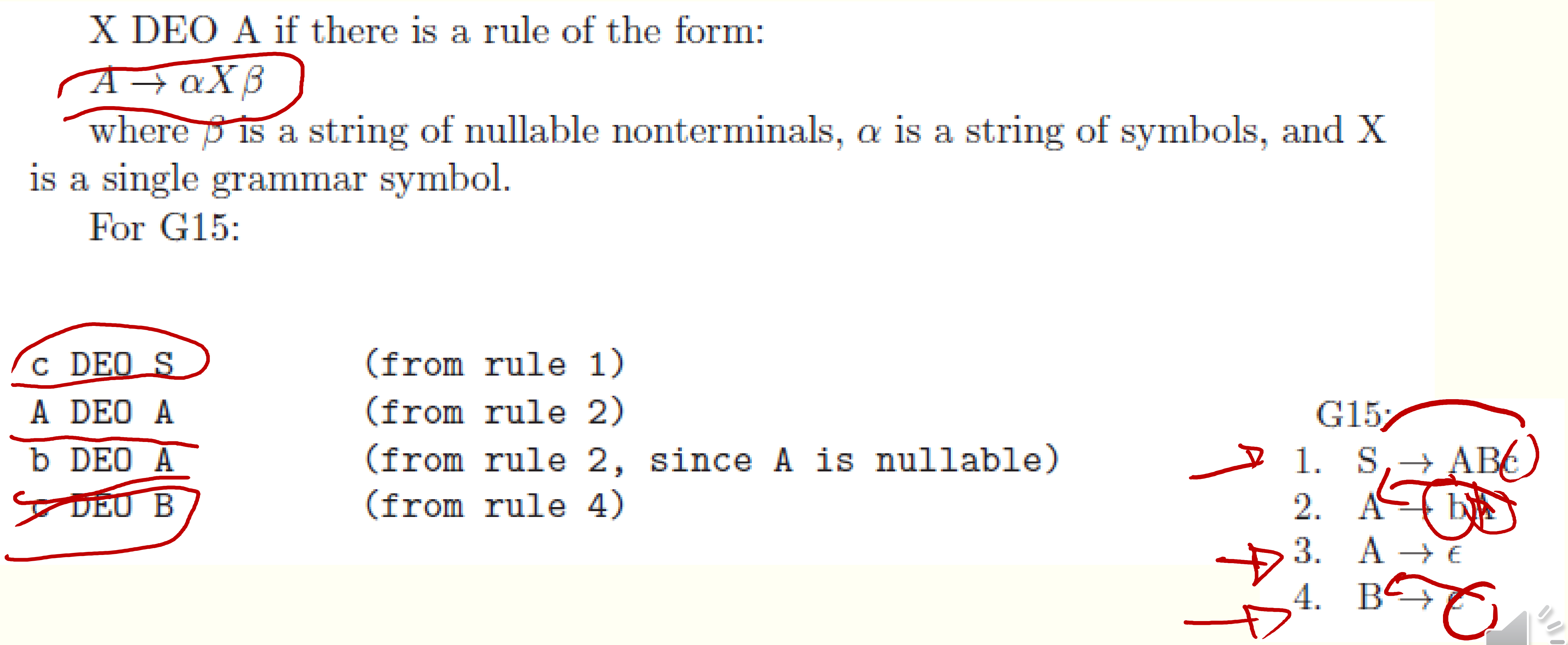
For G15:

G15:
1. S → ABc
2. A → bA
3. A → ϵ
4. B → c

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

Note that if B were a nullable nonterminal we would also have A FDB c.

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.

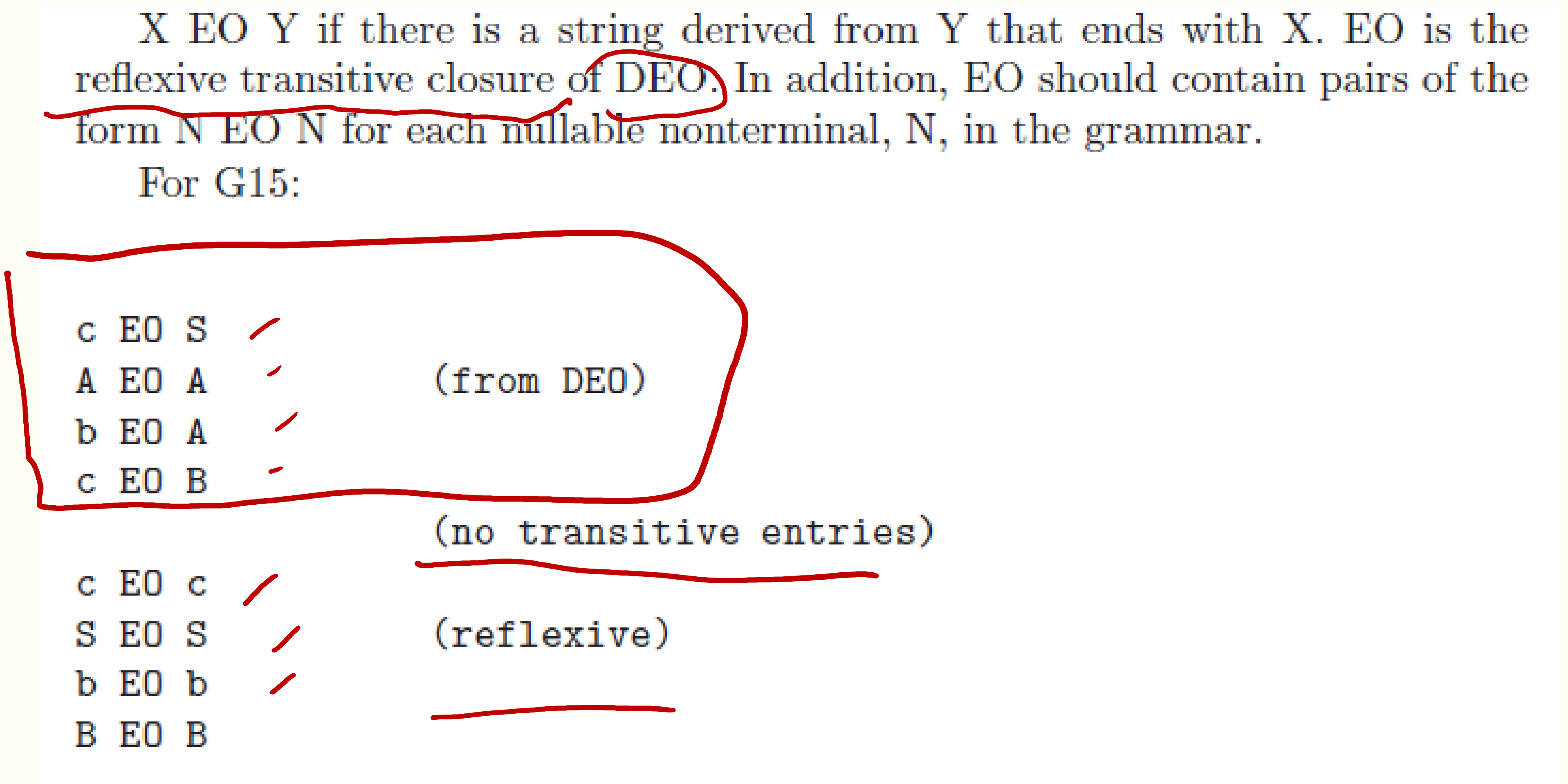
For G15:

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)

G15:

1. S → ABc
2. A → bA
3. A → ε
4. B → c

Step 8: Compute the relation Is End Of



X EO Y if there is a string derived from Y that ends with X. EO is the reflexive transitive closure of DEO. In addition, EO should contain pairs of the form N EO N for each nullable nonterminal, N, in the grammar.

For G15:

```
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

Step 8

```
c EO S
A EO A
b EO A
c EO B

c EO c
S EO S
b EO b
B EO B
```

Step 6

```
A FDB B
B FDB c
```

Step 3

```
S BW A
S BW B
A BW b
B BW c

S BW b
S BW c

S BW S
A BW A
B BW B
b BW b
c BW c
```

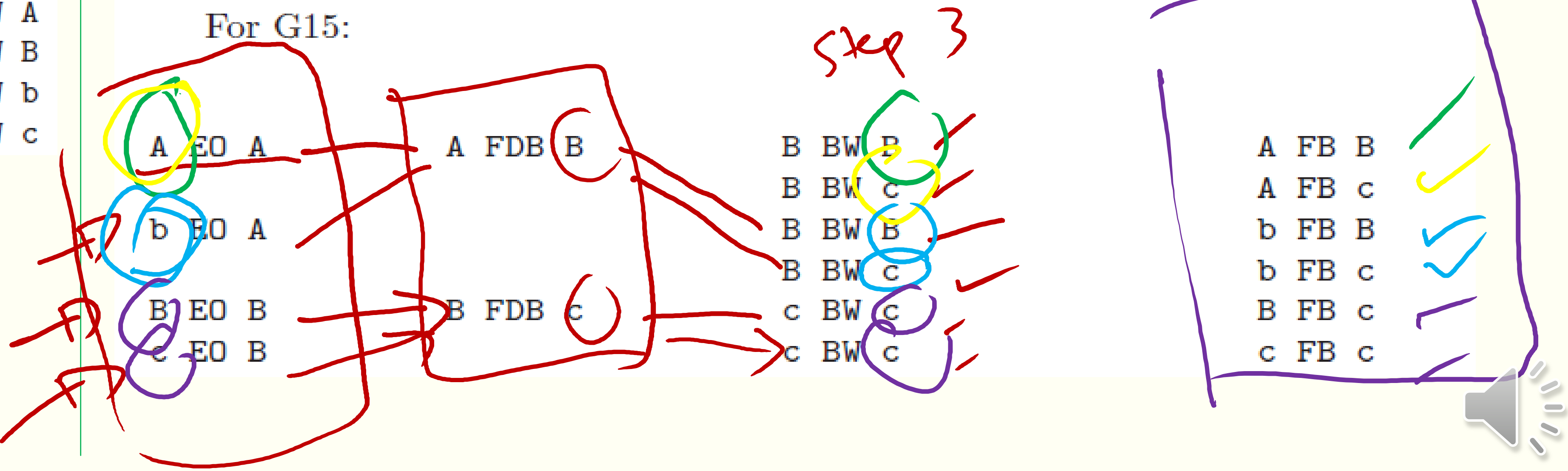
W FB Z if there is a string derived from $S \leftarrow$ in which W is immediately followed by Z.

If there are symbols X and Y such that

```
W EO X
X FDB Y
Y BW Z
```

then W FB Z

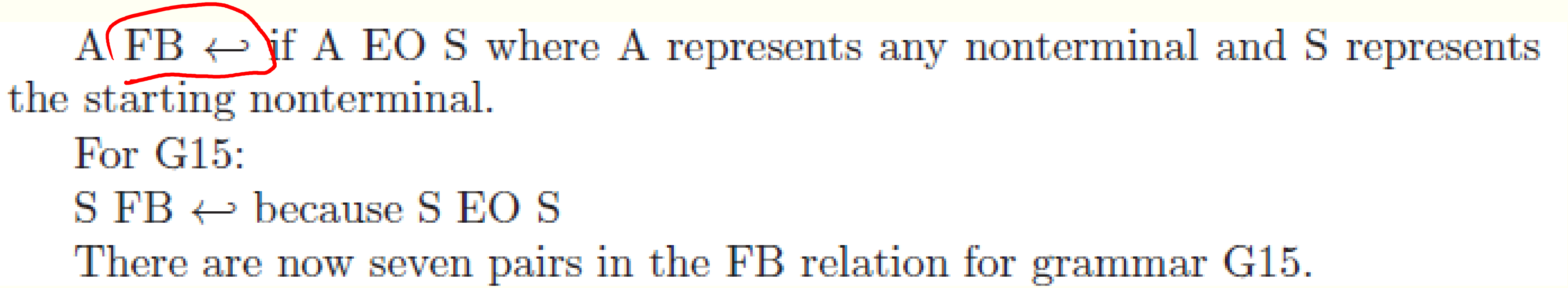
For G15:



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

step 3

Step 10: Extend the FB relation to include end marker



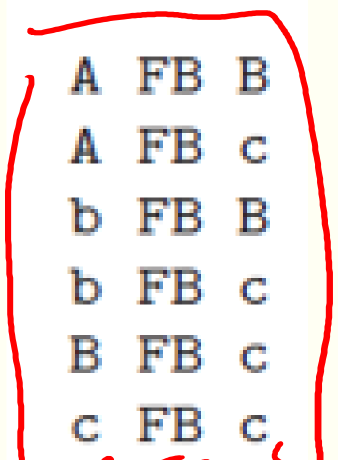
A FB ↩ if A EO S where A represents any nonterminal and S represents the starting nonterminal.

For G15:

S FB ↩ because S EO S

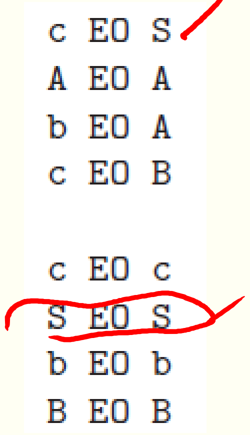
There are now seven pairs in the FB relation for grammar G15.

From step 9



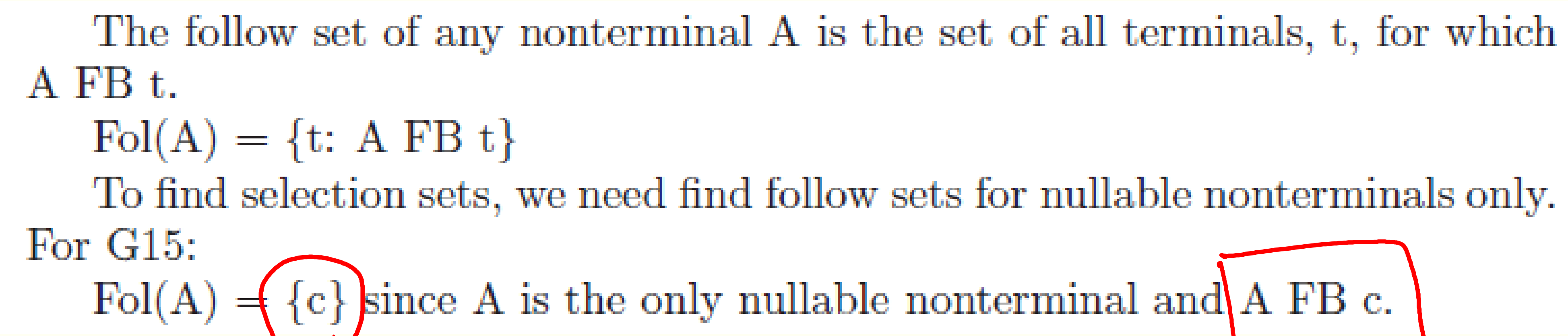
A	FB	B
A	FB	c
b	FB	B
b	FB	c
B	FB	c
c	FB	c

S FB S



c	EO	S
A	EO	A
b	EO	A
c	EO	B
c	EO	c
S	EO	S
b	EO	b
B	EO	B

Step 11: Compute the Follow Set for each nullable nonterminal



The follow set of any nonterminal A is the set of all terminals, t, for which A FB t.

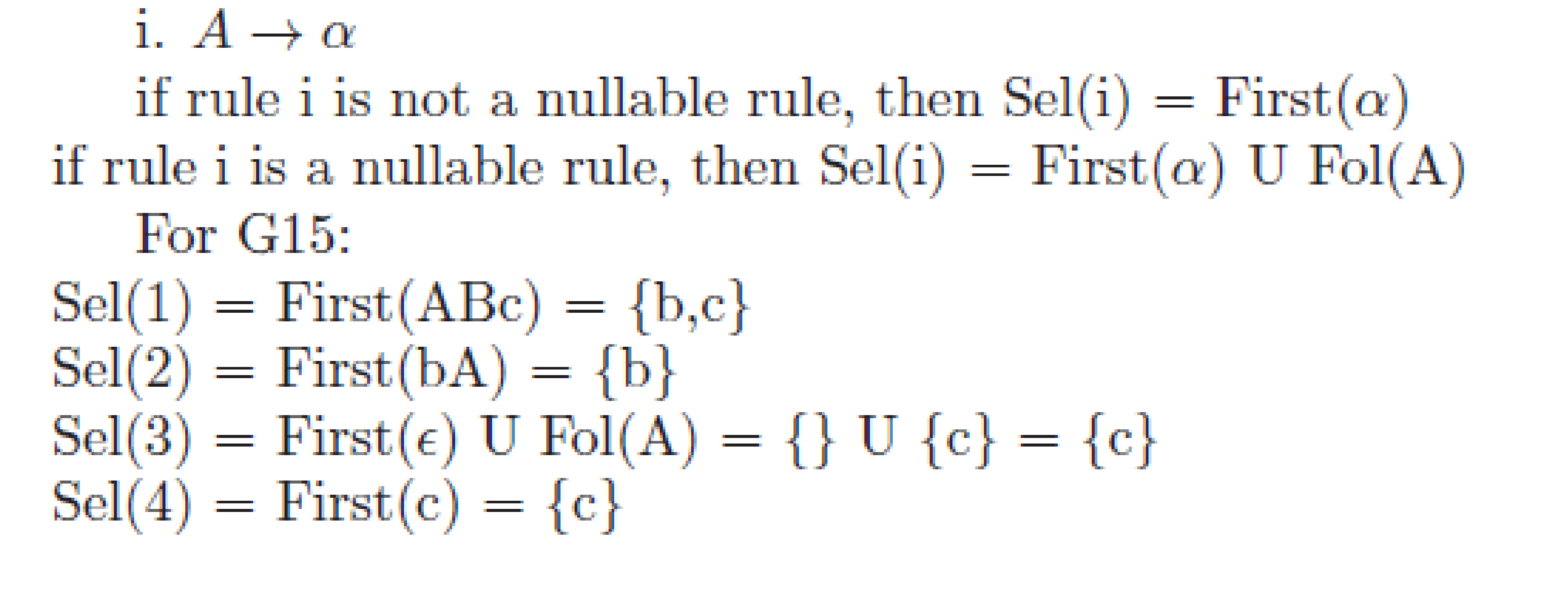
Fol(A) = {t: A FB t}

To find selection sets, we need find follow sets for nullable nonterminals only.

For G15:

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

For G15:

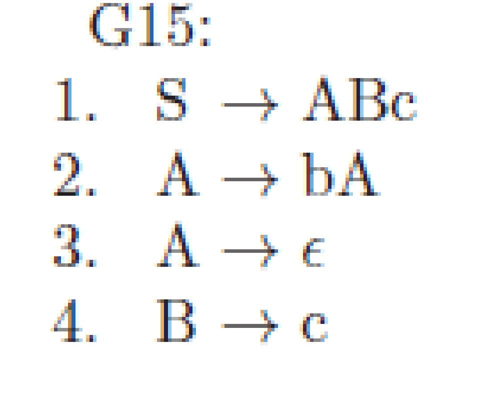
Sel(1) = First(ABc) = {b,c}

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

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

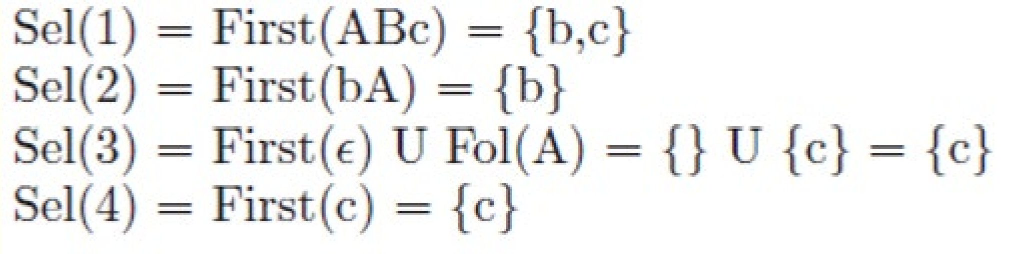
Sel(4) = First(c) = {c}

Pushdown Machines for LL(1) Grammars

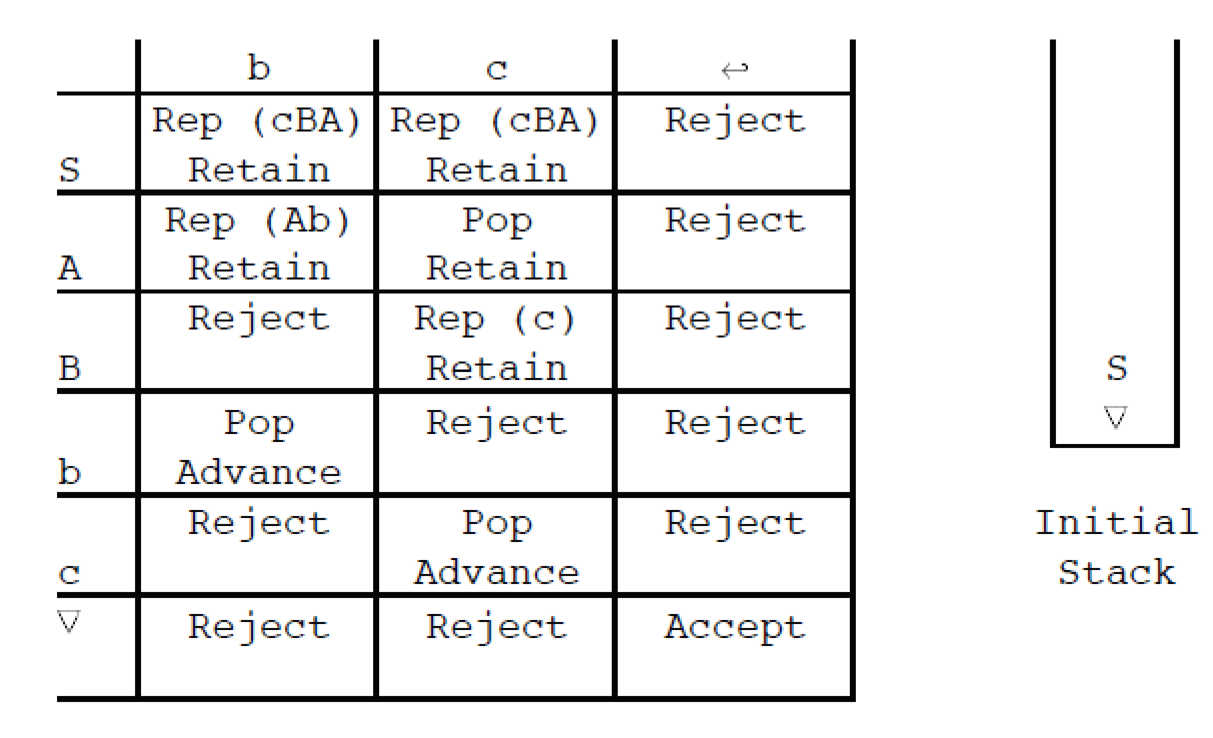


G15:

1. $S \rightarrow ABc$
2. $A \rightarrow bA$
3. $A \rightarrow \epsilon$
4. $B \rightarrow c$



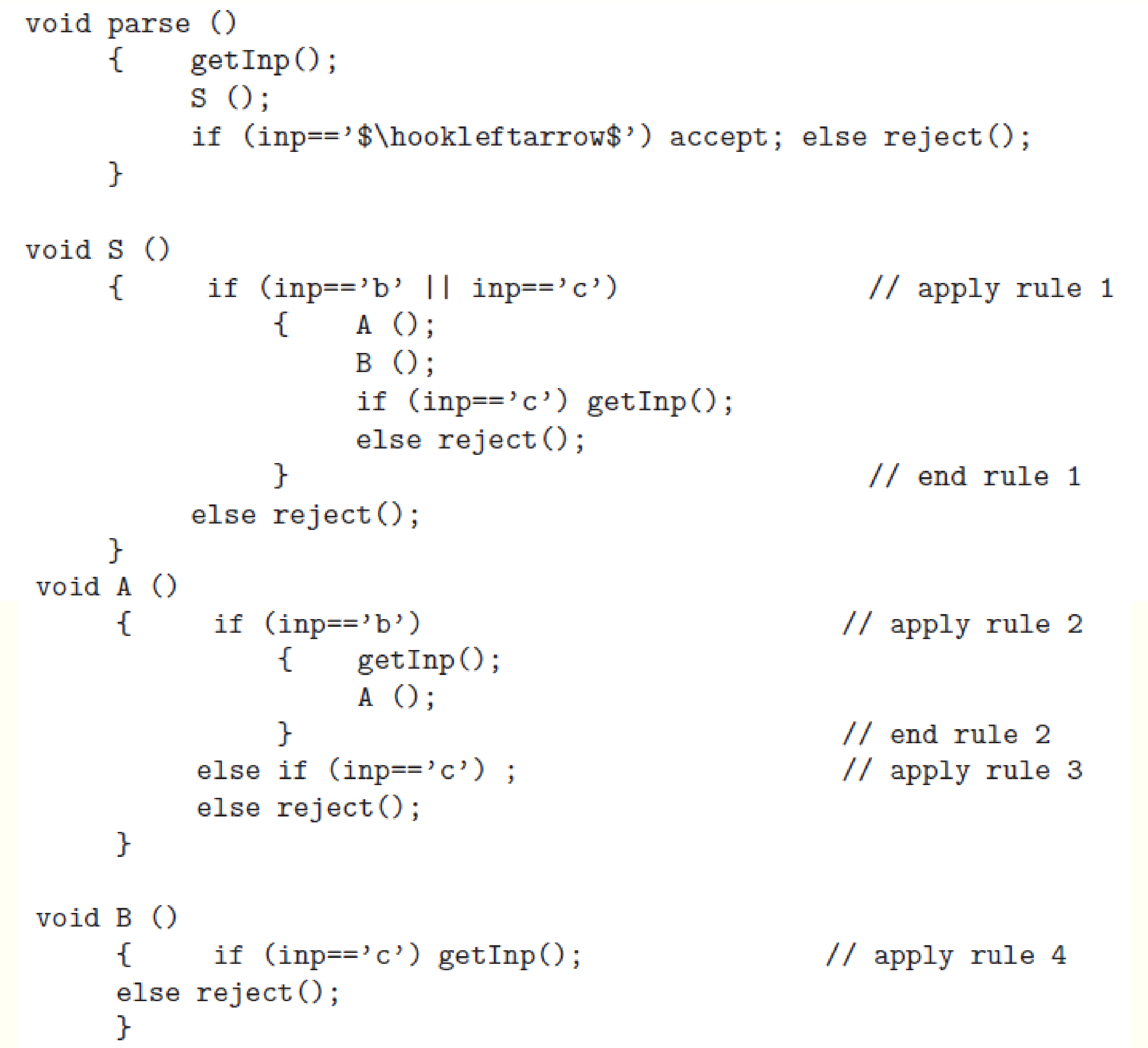
Sel(1) = First(ABc) = {b,c}
Sel(2) = First(bA) = {b}
Sel(3) = First(ϵ) U Fol(A) = {} U {c} = {c}
Sel(4) = First(c) = {c}



	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

Recursive Descent for LL(1) Grammars



```
void parse ()
    {   getInp();
        S ();
        if (inp=='$\hookleftarrow$') accept; else reject();
    }

void S ()
    {    if (inp=='b' || inp=='c')                  // apply rule 1
            {   A ();
                B ();
                if (inp=='c') getInp();
                else reject();
            }                                        // end rule 1
        else reject();
    }
 void A ()
    {    if (inp=='b')                            // apply rule 2
            {   getInp();
                A ();
            }                                     // end rule 2
        else if (inp=='c') ;                      // apply rule 3
        else reject();
    }

 void B ()
    {    if (inp=='c') getInp();                 // apply rule 4
    else reject();
    }
```

END OF LL(1) GRAMMARS

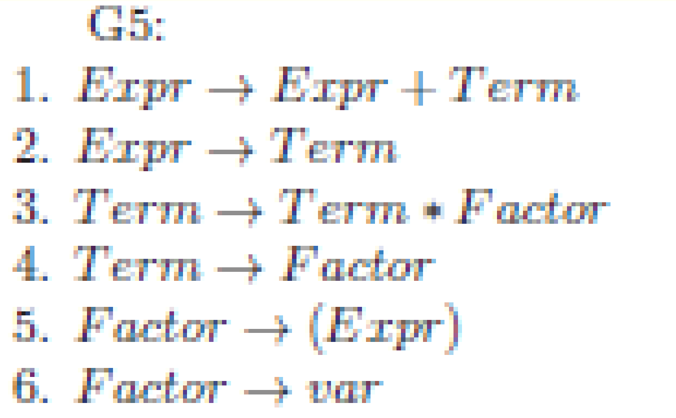
Next: Chapter 5

PARSING ARITHMETIC EXPRESSIONS TOP DOWN

Introduction

- One of the most heavily studied aspects of parsing programming languages deals with arithmetic expressions.

- Consider Grammar G15, we need to determine whether it is in LL(1) or not?



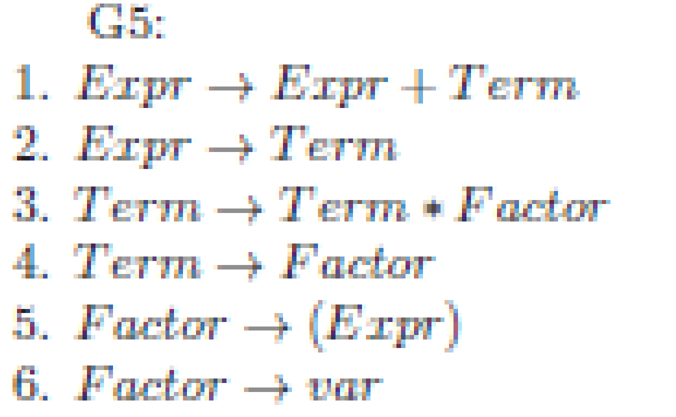
G5:

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

- In order to determine whether this grammar is LL(1), we must first find the selection set for each rule in the grammar.

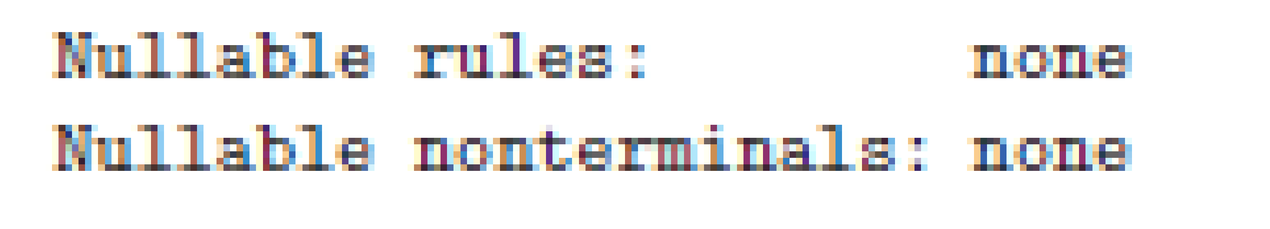
- We do this by using the twelve-step algorithm.

Step 1



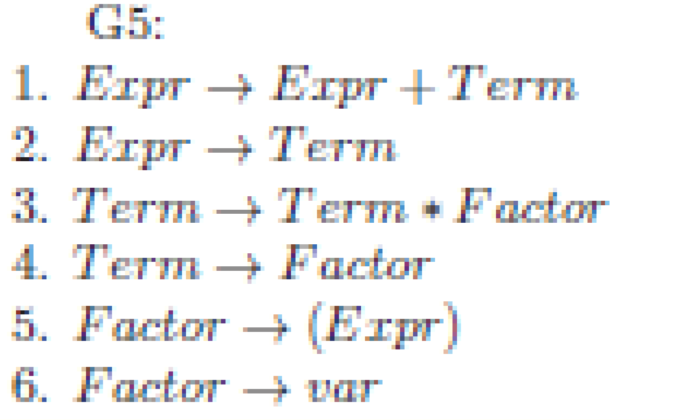
G5:

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



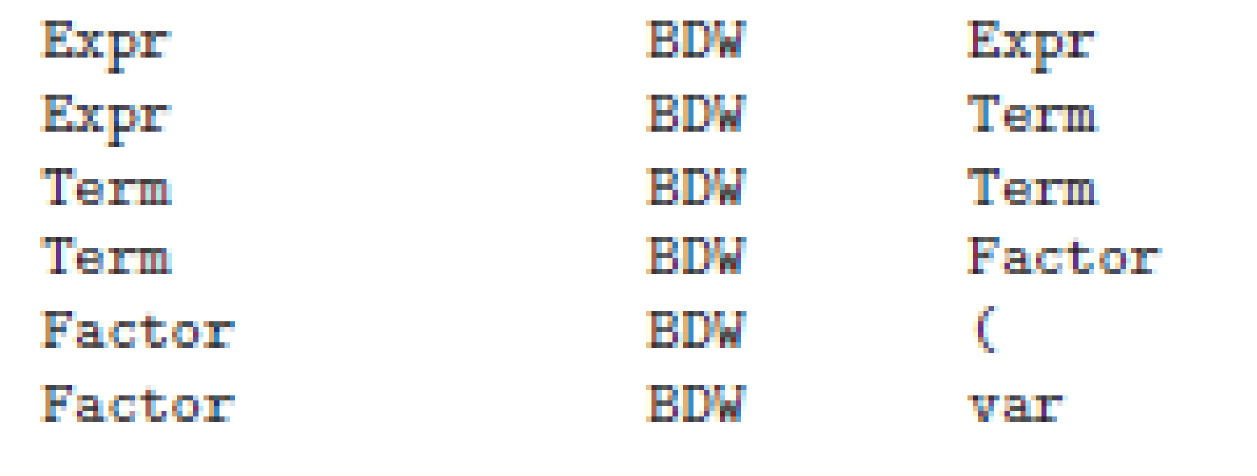
```
Nullable rules:         none
Nullable nonterminals:  none
```

Step 2: Find Begin Directly With



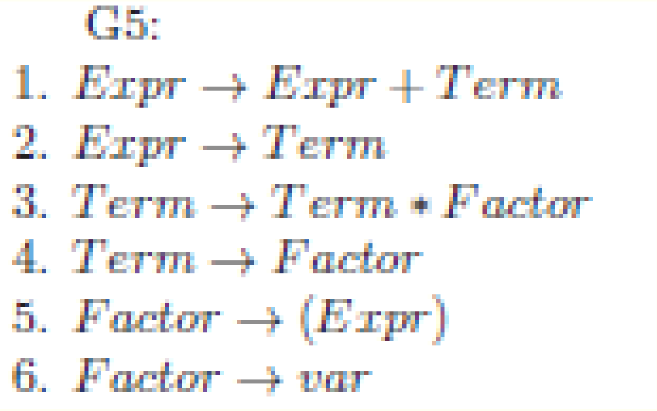
G5:

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



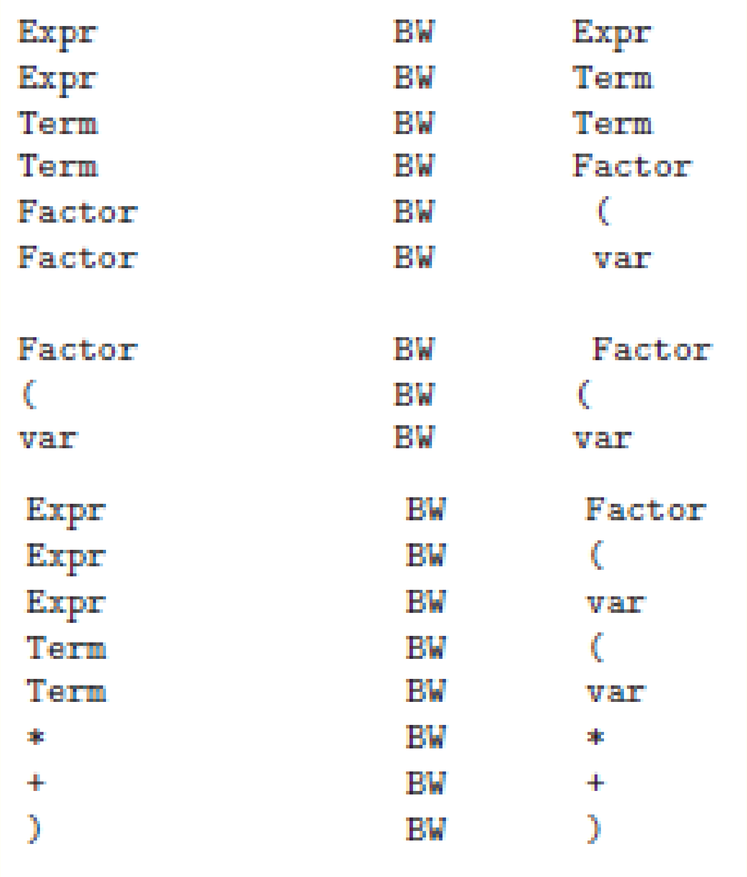
Expr	BDW	Expr
Expr	BDW	Term
Term	BDW	Term
Term	BDW	Factor
Factor	BDW	(
Factor	BDW	var

Step 3: Find Begin With (Reflexive transitive Closure)



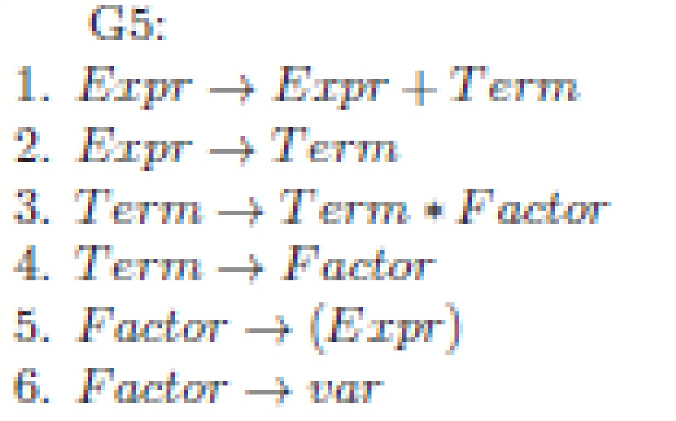
G5:

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



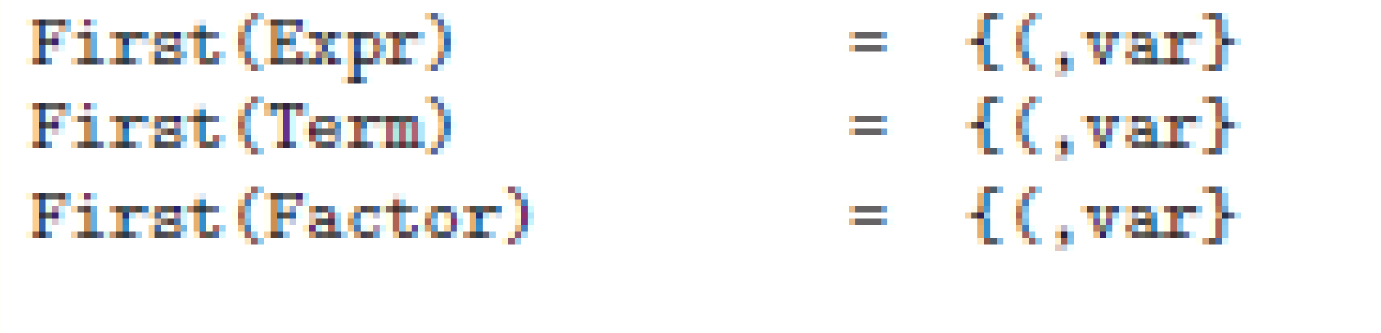
Expr	BW	Expr
Expr	BW	Term
Term	BW	Term
Term	BW	Factor
Factor	BW	(
Factor	BW	var
Factor	BW	Factor
(	BW	(
var	BW	var
Expr	BW	Factor
Expr	BW	(
Expr	BW	var
Term	BW	(
Term	BW	var
*	BW	*
+	BW	+
)	BW	)

Step 4: Find First(x)



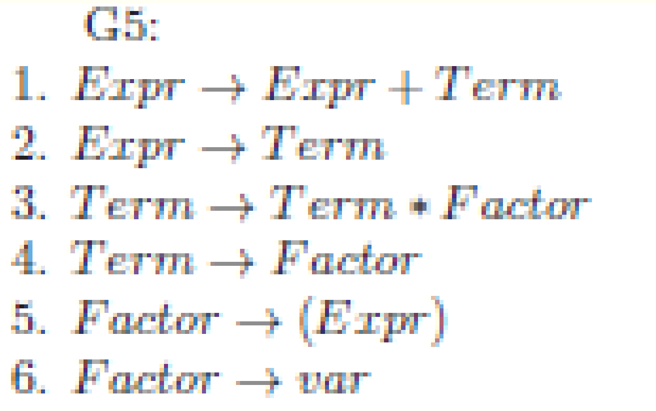
G5:

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



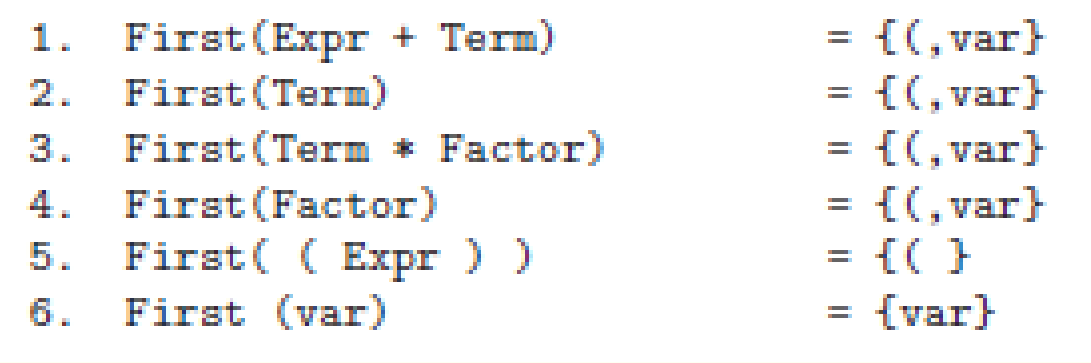
```
First(Expr)            =   {(,var}
First(Term)            =   {(,var}
First(Factor)          =   {(,var}
```

Step 5



G5:

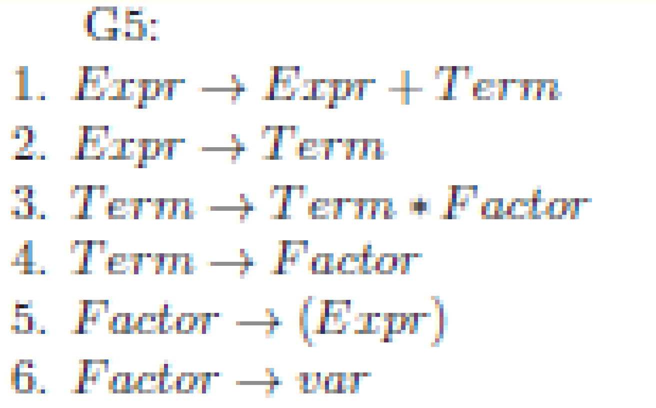
1. $Expr \rightarrow Expr + Term$
2. $Expr \rightarrow Term$
3. $Term \rightarrow Term * Factor$
4. $Term \rightarrow Factor$
5. $Factor \rightarrow (Expr)$
6. $Factor \rightarrow var$



```
1.   First(Expr + Term)          = {(,var}
2.   First(Term)                 = {(,var}
3.   First(Term * Factor)        = {(,var}
4.   First(Factor)               = {(,var}
5.   First( ( Expr ) )           = {( }
6.   First (var)                 = {var}
```

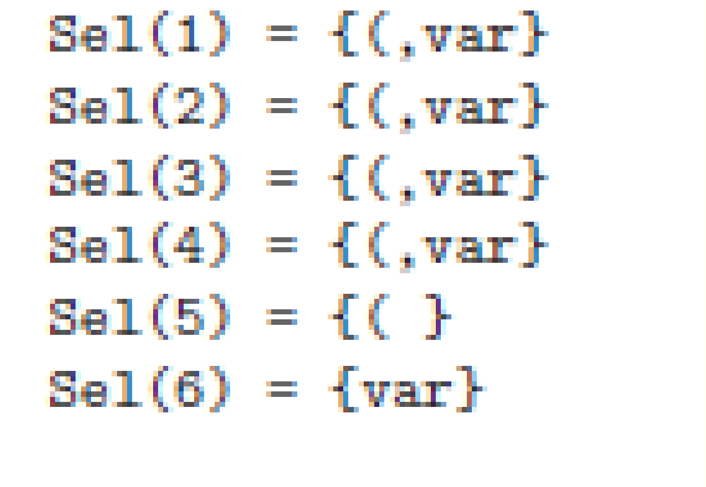
Since there are no nullable rules in the grammar, we can obtain the selection sets directly from step 5.

Jump to step 12



G5:

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



```
Sel(1) = {(,var}
Sel(2) = {(,var}
Sel(3) = {(,var}
Sel(4) = {(,var}
Sel(5) = {( }
Sel(6) = {var}
```

BUT, this grammar is not LL(1) because

rules 1 and 2 define the same nonterminal, Expr, and their selection sets intersect.

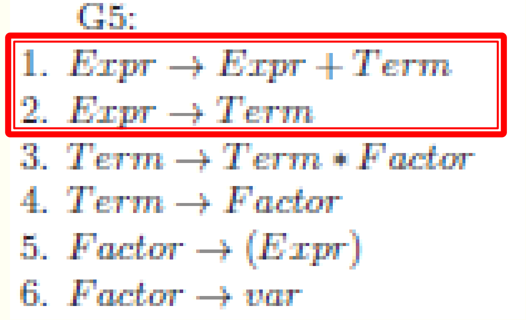
This is also true for rules 3 and 4.

Grammar G5 is not suitable for top down parsing, it can be determined much more easily by inspection of the grammar. Rules 1 and 3 both have a property known as left recursion:

1. Expr → Expr + T erm
3. T erm → T erm∗ Factor

Any grammar with left recursion cannot be LL(1)

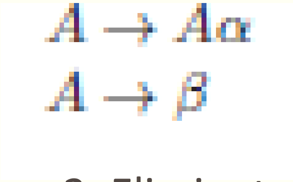
Eliminating Left Recursion



G5:

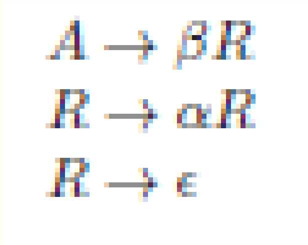
1. $Expr \rightarrow Expr + Term$
2. $Expr \rightarrow Term$
3. $Term \rightarrow Term * Factor$
4. $Term \rightarrow Factor$
5. $Factor \rightarrow (Expr)$
6. $Factor \rightarrow var$

1. General rule



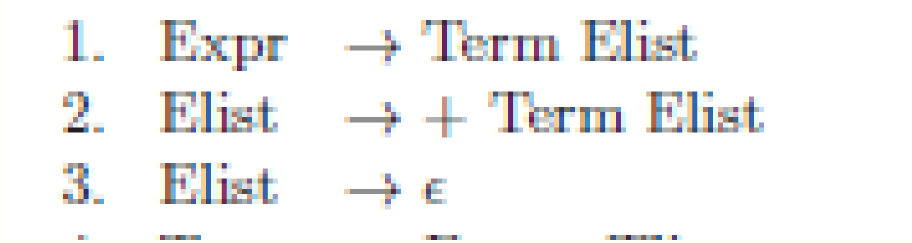
$$A \rightarrow A\alpha$$
$$A \rightarrow \beta$$

2. Eliminate the left recursion by introducing a new nonterminal, R, and rewriting the rules as:



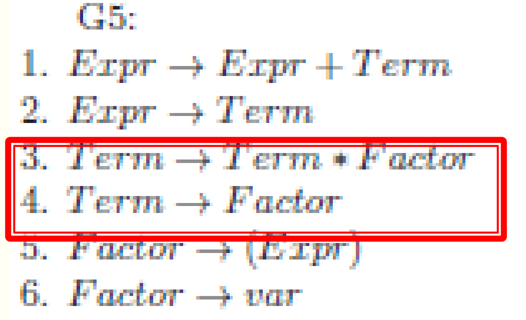
$$A \rightarrow \beta R$$
$$R \rightarrow \alpha R$$
$$R \rightarrow \epsilon$$

3. New rule after eliminating left recursion with new nonterminal Elist



1. Expr $\rightarrow$ Term Elist
2. Elist $\rightarrow$ + Term Elist
3. Elist $\rightarrow$ ϵ

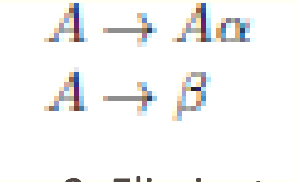
Eliminating Left Recursion



G5:

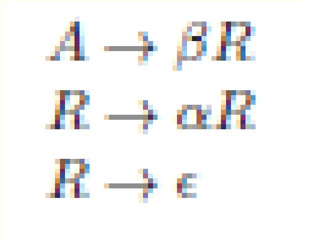
1. $Expr \to Expr + Term$
2. $Expr \to Term$
3. $Term \to Term * Factor$
4. $Term \to Factor$
5. $Factor \to (Expr)$
6. $Factor \to var$

1. General rule



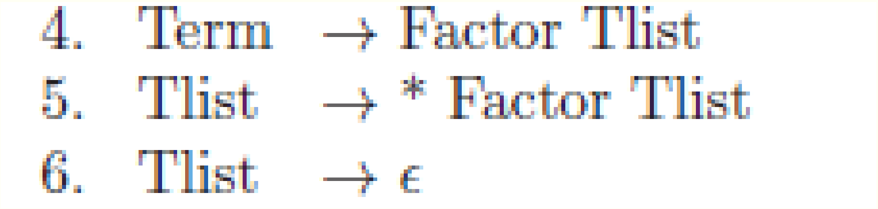
$$A \to A\alpha$$
$$A \to \beta$$

2. Eliminate the left recursion by introducing a new nonterminal, R, and rewriting the rules as:



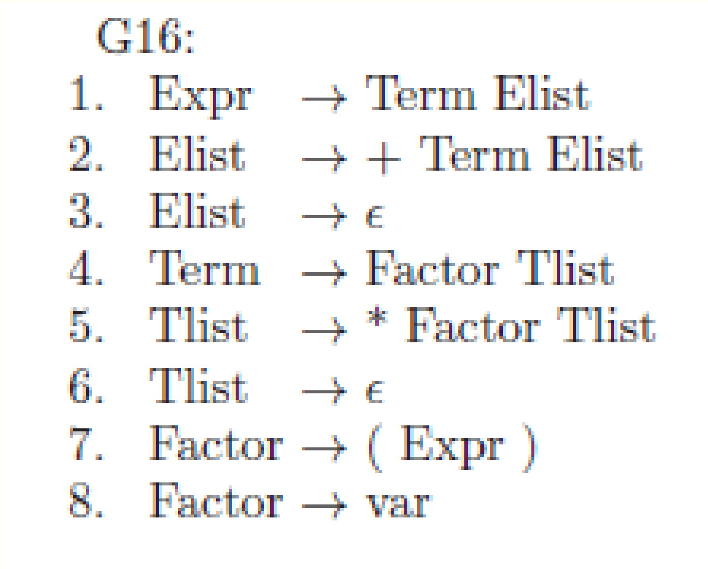
$$A \to \beta R$$
$$R \to \alpha R$$
$$R \to \epsilon$$

3. New rule after eliminating left recursion with new nonterminal Tlist



4. Term $\to$ Factor Tlist
5. Tlist $\to$ * Factor Tlist
6. Tlist $\to$ ϵ

New G16 grammar



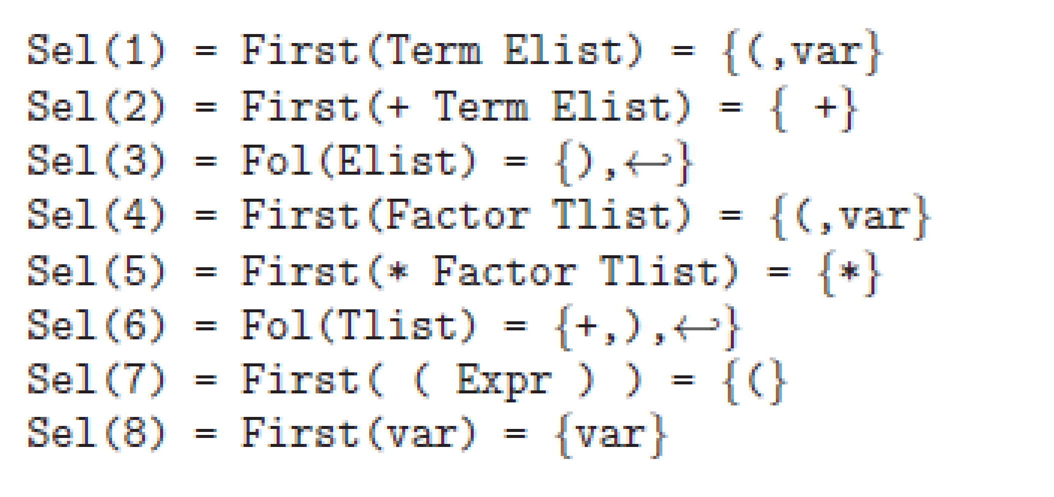
G16:

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

NEXT:

Check whether G16 is in LL(1) or not, using twelve-step algorithm
(can refer to previous video on LL(1) part 1 and LL(1) part 2)

Step 12 output of G16

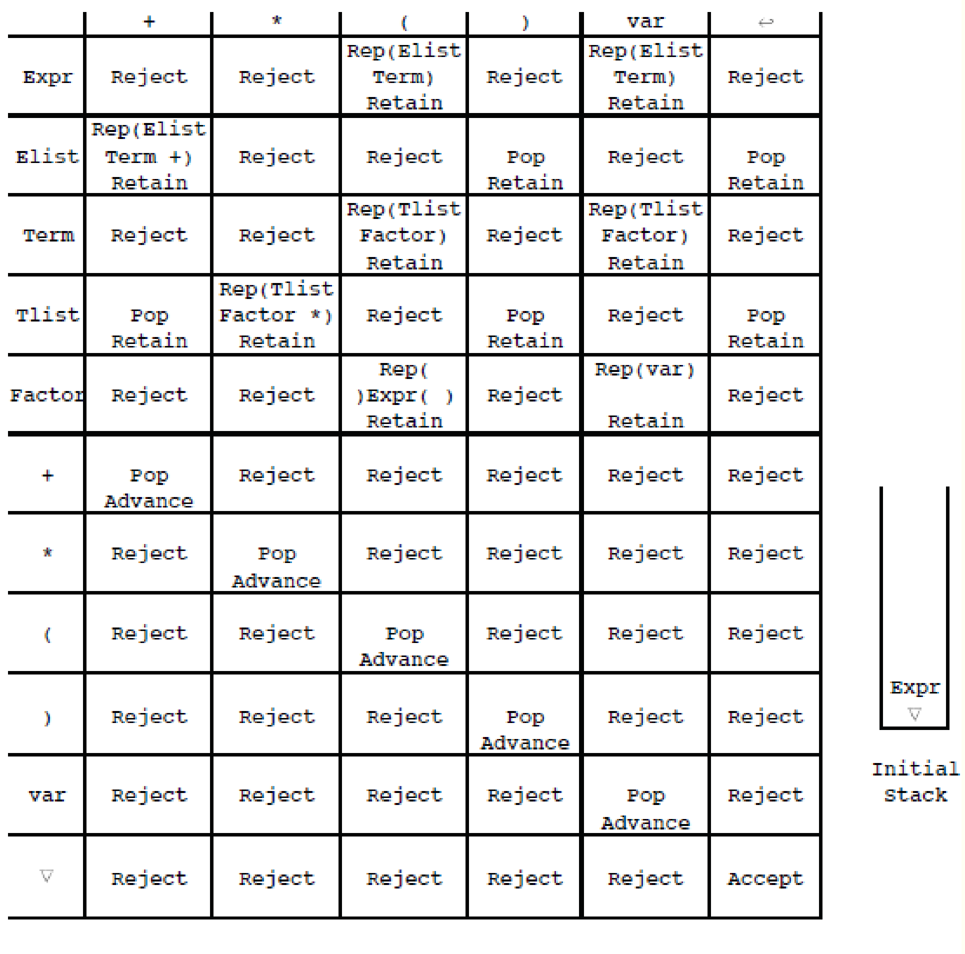


```
Sel(1) = First(Term Elist) = {(,var}
Sel(2) = First(+ Term Elist) = { +}
Sel(3) = Fol(Elist) = {),↩}
Sel(4) = First(Factor Tlist) = {(,var}
Sel(5) = First(* Factor Tlist) = {*}
Sel(6) = Fol(Tlist) = {+,),↩}
Sel(7) = First( ( Expr ) ) = {(}
Sel(8) = First(var) = {var}
```

Since all rules defining the same nonterminal (rules 2 and 3, rules 5 and 6, rules 7 and 8) have disjoint selection sets, the grammar G16 is LL(1).

Extended Pushdown Machine

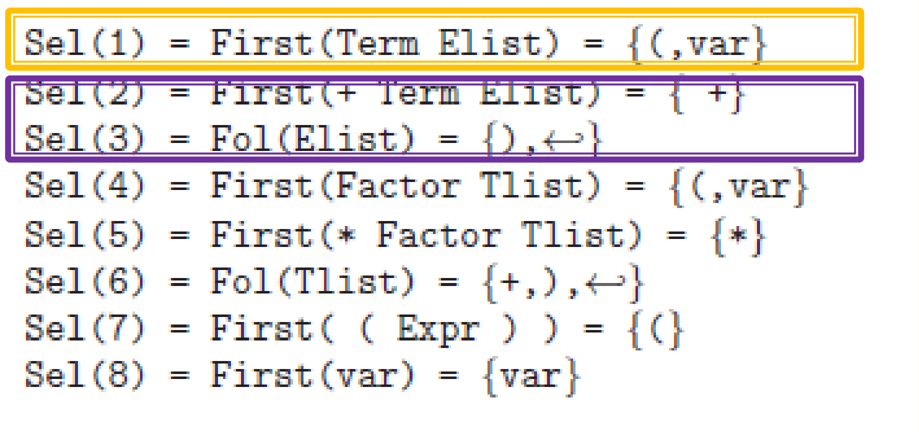
- Pushdown machine is constructed based on the selection obtained in step 12.
- For example, since the selection set for rule 4 is {(,var}, we fill the cells in the row labeled Term and columns labeled (and var with information from rule 4: Rep (Tlist Factor)



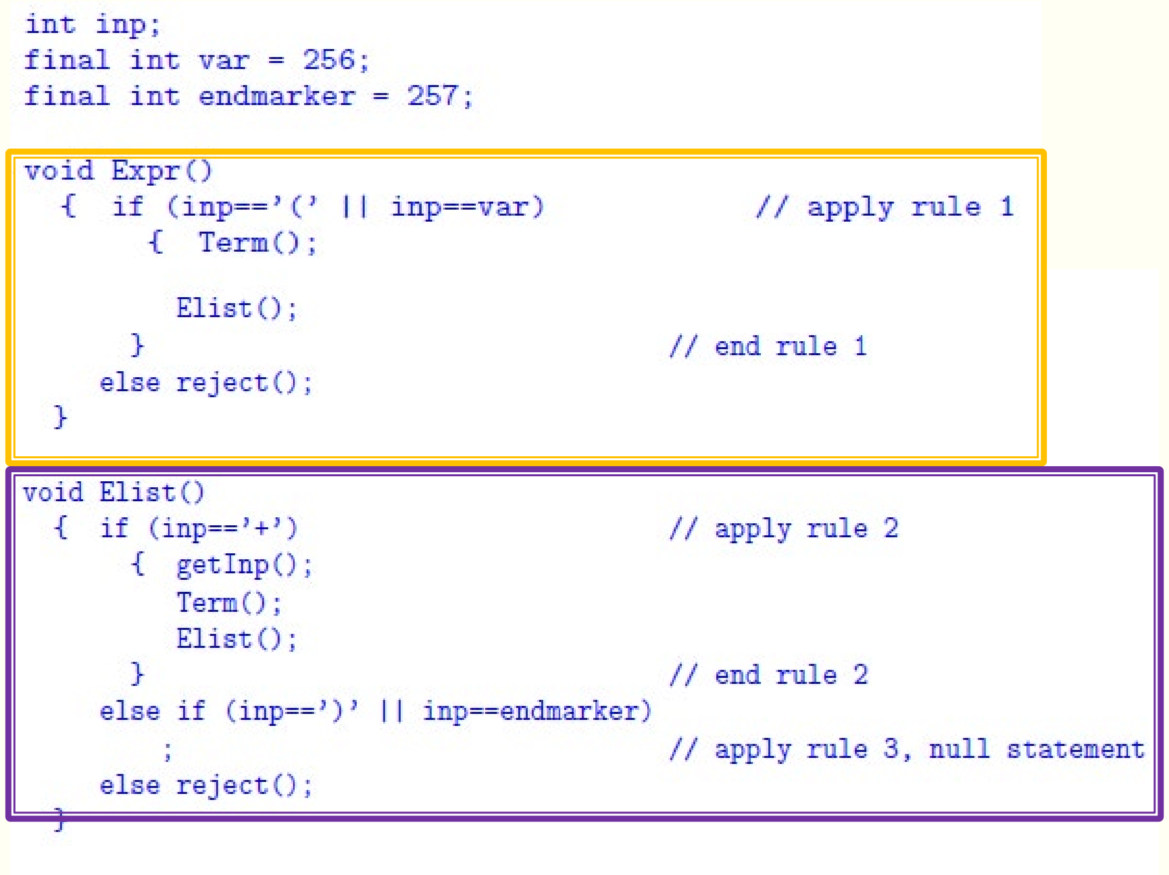
	+	*	(	)	var	↩
Expr	Reject	Reject	Rep(Elist Term) Retain	Reject	Rep(Elist Term) Retain	Reject
Elist	Rep(Elist Term +) Retain	Reject	Reject	Pop Retain	Reject	Pop Retain
Term	Reject	Reject	Rep(Tlist Factor) Retain	Reject	Rep(Tlist Factor) Retain	Reject
Tlist	Pop Retain	Rep(Tlist Factor *) Retain	Reject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

Initial Stack

Recursive Descent Parser

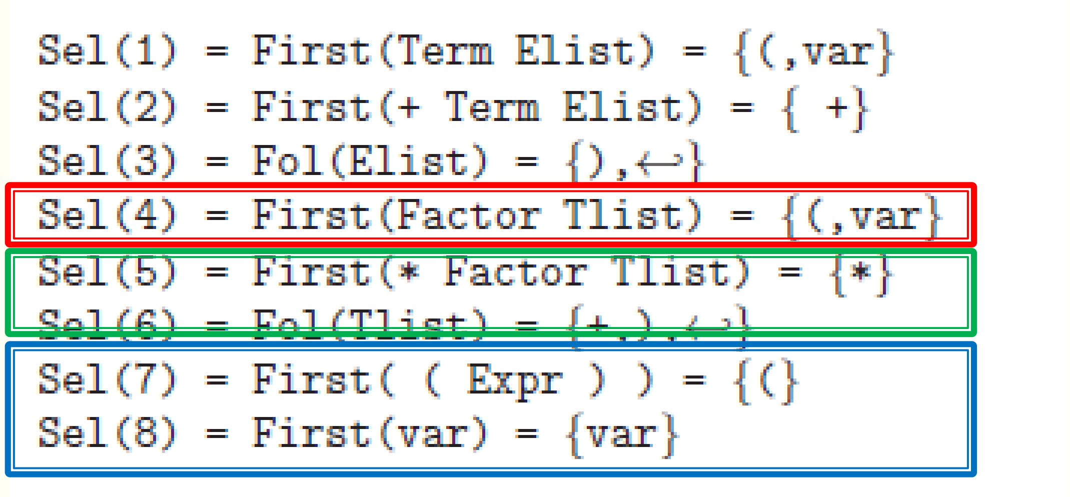


```
Sel(1) = First(Term Elist) = {(,var}
Sel(2) = First(+ Term Elist) = { +}
Sel(3) = Fol(Elist) = {),↩}
Sel(4) = First(Factor Tlist) = {(,var}
Sel(5) = First(* Factor Tlist) = {*}
Sel(6) = Fol(Tlist) = {+,),↩}
Sel(7) = First( ( Expr ) ) = {(}
Sel(8) = First(var) = {var}
```

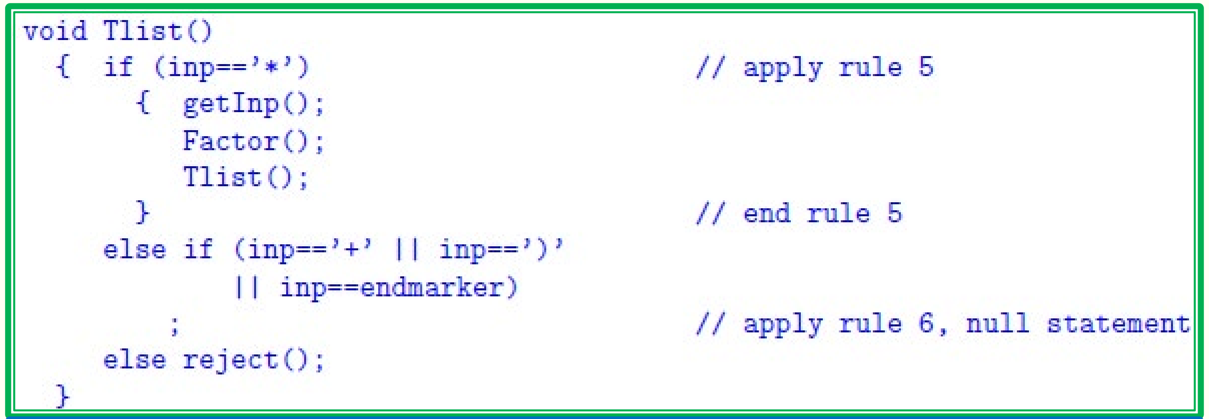


```
int inp;
final int var = 256;
final int endmarker = 257;

void Expr()
 {  if (inp=='(' || inp==var)              // apply rule 1
      {  Term();

         Elist();
     }                                    // end rule 1
    else reject();
 }

void Elist()
 {  if (inp=='+')                         // apply rule 2
     {  getInp();
        Term();
        Elist();
     }                                    // end rule 2
    else if (inp==')' || inp==endmarker)
       ;                                  // apply rule 3, null statement
    else reject();
 }
```

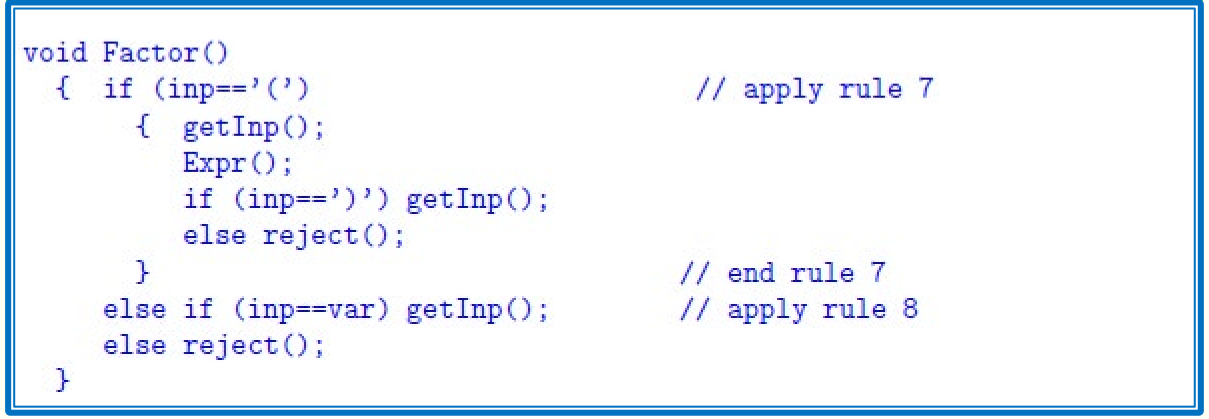
Cont: Recursive Descent Parser



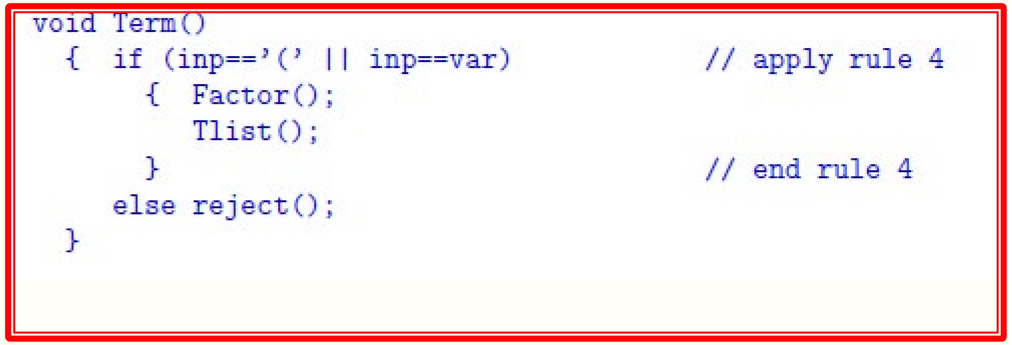
```
Sel(1) = First(Term Elist) = {(,var}
Sel(2) = First(+ Term Elist) = { +}
Sel(3) = Fol(Elist) = {),↩}
Sel(4) = First(Factor Tlist) = {(,var}
Sel(5) = First(* Factor Tlist) = {*}
Sel(6) = Fol(Tlist) = {+,),↩}
Sel(7) = First( ( Expr ) ) = {(}
Sel(8) = First(var) = {var}
```



```
void Tlist()
  {  if (inp=='*')                         // apply rule 5
       {  getInp();
          Factor();
          Tlist();
       }                                   // end rule 5
     else if (inp=='+' || inp==')'
             || inp==endmarker)
         ;                                 // apply rule 6, null statement
     else reject();
  }
```



```
void Factor()
  {  if (inp=='(')                         // apply rule 7
       {  getInp();
          Expr();
          if (inp==')') getInp();
          else reject();
       }                                  // end rule 7
     else if (inp==var) getInp();         // apply rule 8
     else reject();
  }
```



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

END OF PARSING ARITHMETIC EXPRESSIONS TOP DOWN

Next: Chapter 5

SYNTAX-DIRECTED TRANSLATION

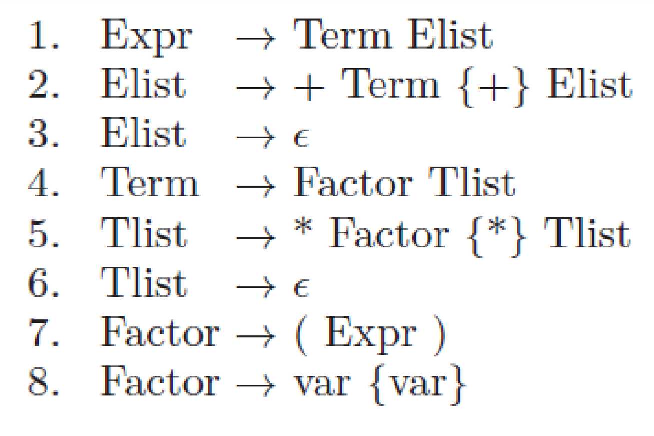
Introduction

- The top down parsing in simple, quasi-simple and LL(1) grammar can only check for syntax errors.
- they cannot produce output, and they do not deal at all with semantics (the intent or meaning) of the source program.
- For this purpose, we now introduce **action symbols** which are intended to give us
  - the capability of producing output and/or
  - calling other methods while parsing an input string
- A grammar containing action symbols is called a translation grammar

Translation grammar

- action symbols in a grammar is represented using curly braces {}.
- the action symbol purpose is to produce output.
- If you need to find the selection set for translation grammar, you must remove the action symbol first.
- The resulted grammar is called underlying grammar.

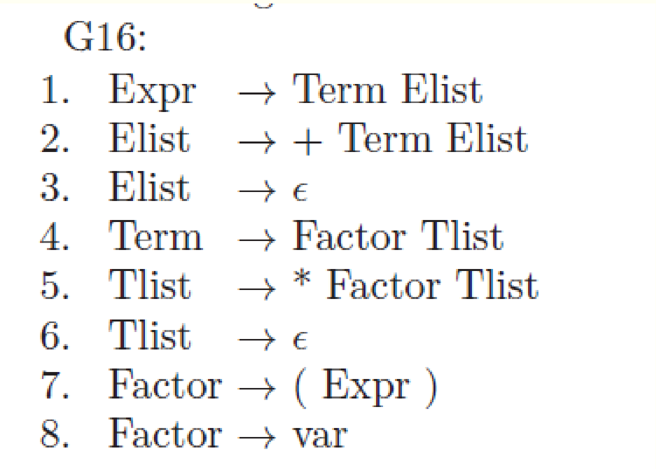
Grammar G17



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}

Translation grammar

remove action symbol



G16:

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

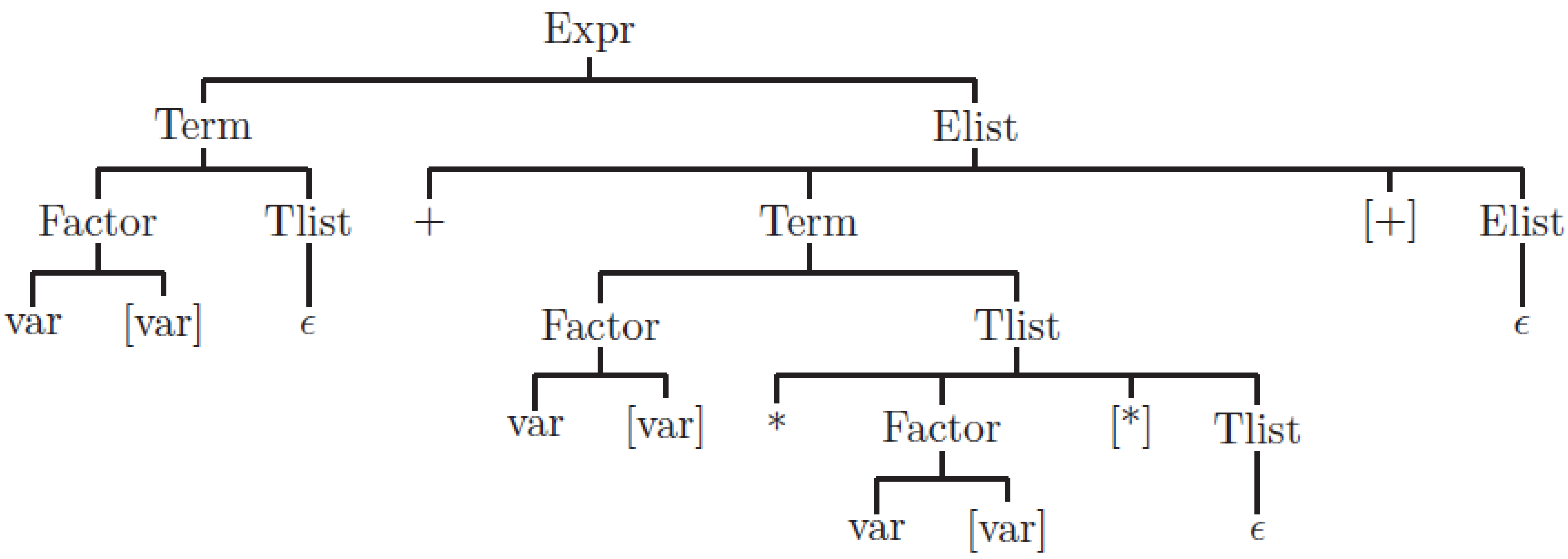
Underlying grammar

Example

- G17 is a translation grammar to translate infix expressions involving addition and multiplication to postfix

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}

Postfix output: {var} {var} {var} {*} {+}



A derivation tree for the expression var+var*var using grammar G17

Implementing Translation Grammars with Pushdown Translators

action symbols should be treated as stack symbols

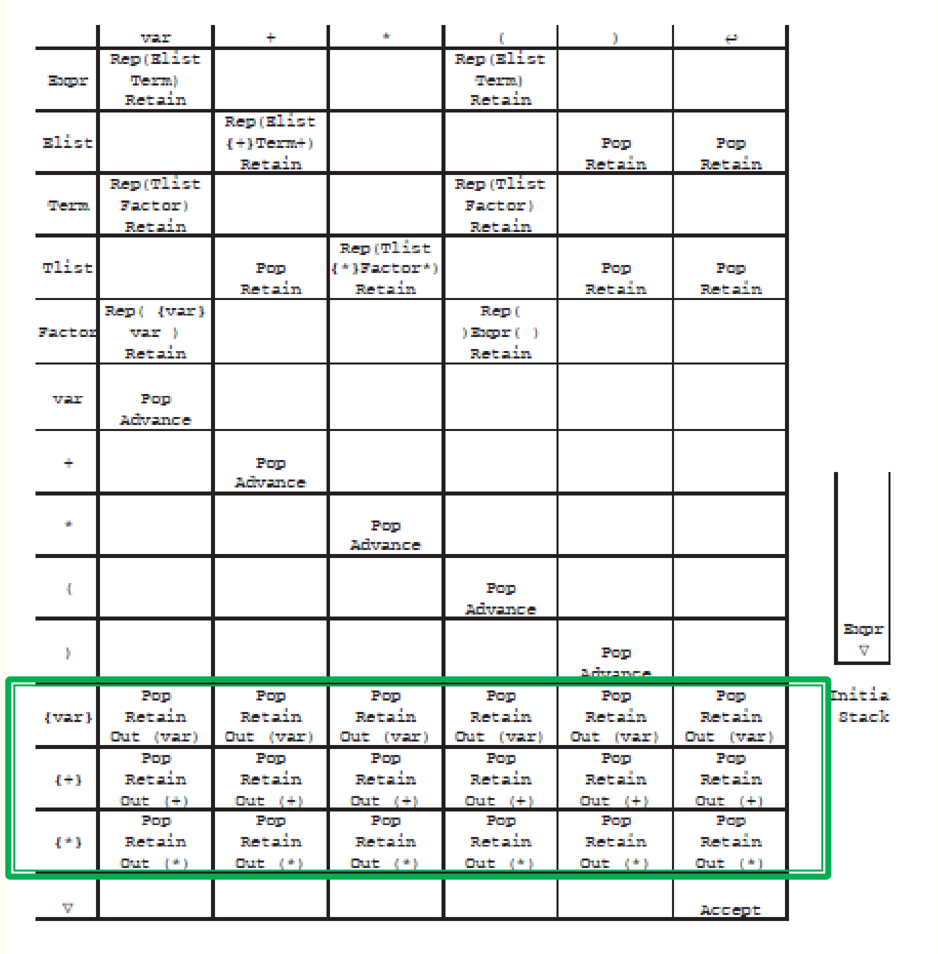
Thus, each action symbol {A} representing output should label a row of the pushdown machine table.

Every column of that row should contain the entry Out(A), Pop, Retain

The pushdown machine is then called, extended pushdown machine

Extended Pushdown Machine for G17

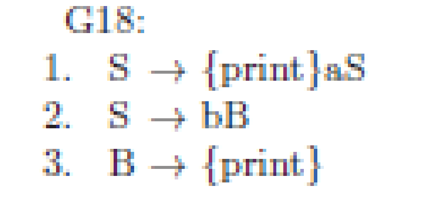
1. Expr → Term Elist
2. Elist → + Term {+} Elist
3. Elist → ϵ
4. Term → Factor Tlist
5. Tlist → * Factor {*} Tlist
6. Tlist → ϵ
7. Factor → (Expr)
8. Factor → var {var}



	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

Implementing Translation Grammars with Recursive Descent

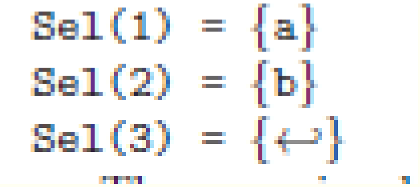
- Grammar



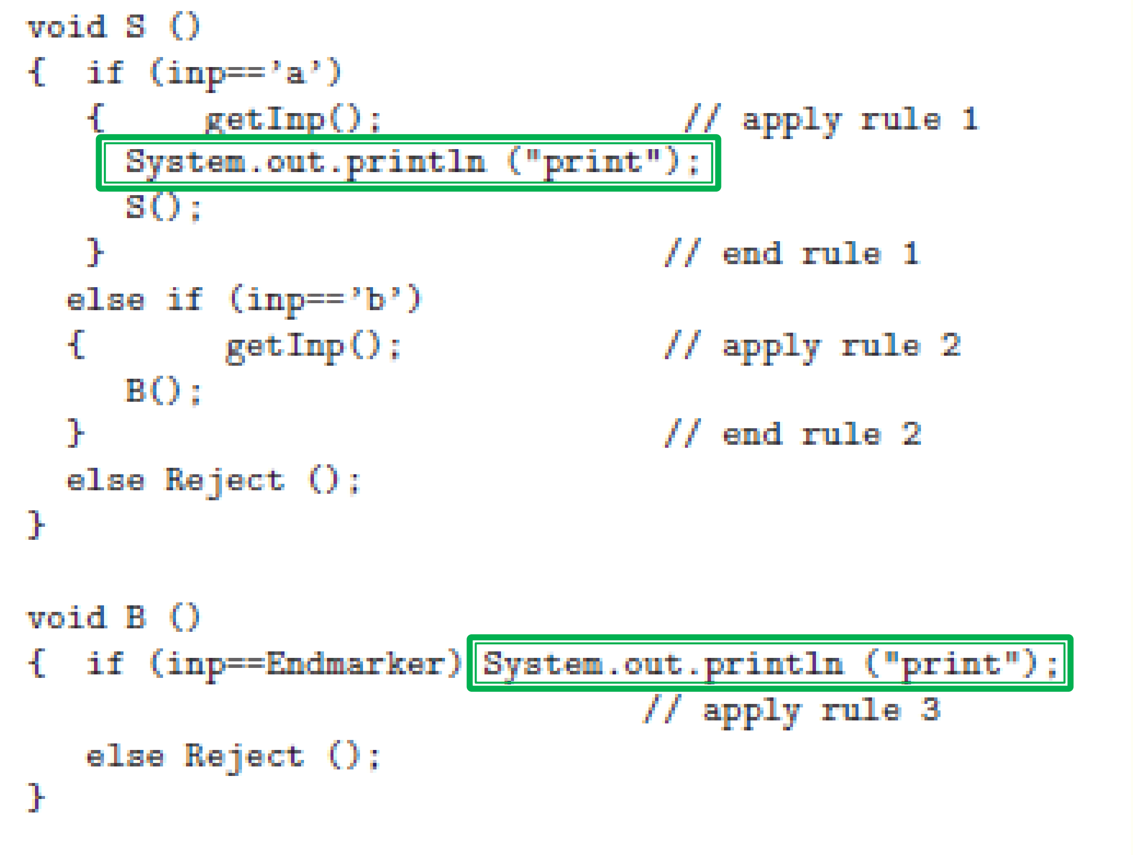
G18:

1. S → {print}aS
2. S → bB
3. B → {print}

- Selection set



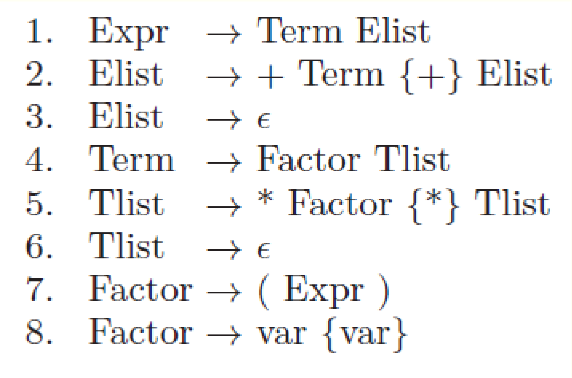
Sel(1) = {a}
Sel(2) = {b}
Sel(3) = {↩}



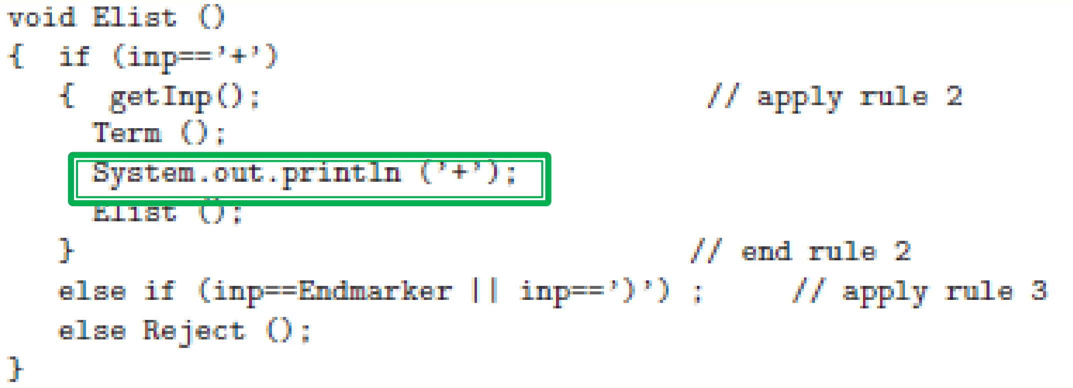
```
void S ()
{  if (inp==’a’)
   {     getInp();                 // apply rule 1
     System.out.println ("print");
     S();
   }                              // end rule 1
  else if (inp==’b’)
  {       getInp();               // apply rule 2
     B();
  }                               // end rule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

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

Example: The recursive descent translator for grammar G17

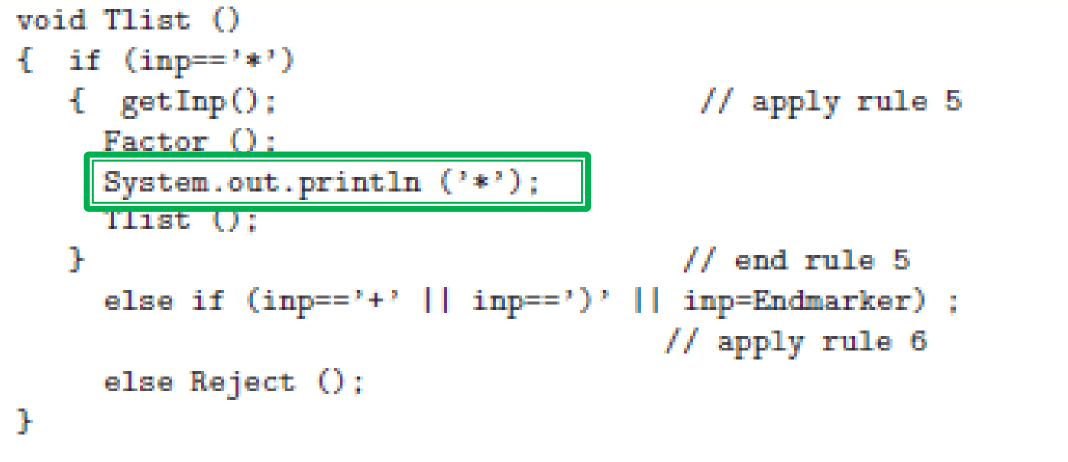
G17



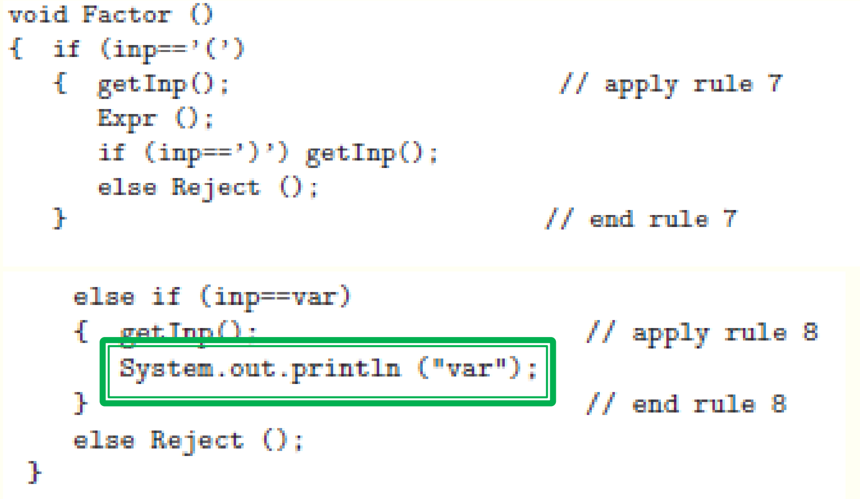
1. Expr → Term Elist
2. Elist → + Term {+} Elist
3. Elist → ϵ
4. Term → Factor Tlist
5. Tlist → * Factor {*} Tlist
6. Tlist → ϵ
7. Factor → (Expr)
8. Factor → var {var}



```
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 ();
}
```



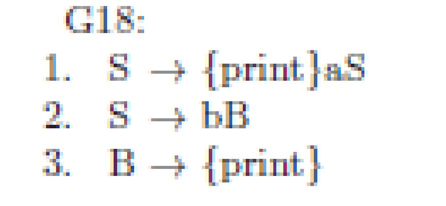
```
void Tlist ()
{  if (inp=='*')
   {  getInp();                           // apply rule 5
      Factor ();
      System.out.println ('*');
      Tlist ();
   }                                      // end rule 5
      else if (inp=='+' || inp==')' || inp=Endmarker) ;
                              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



```
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 ();
 }
```

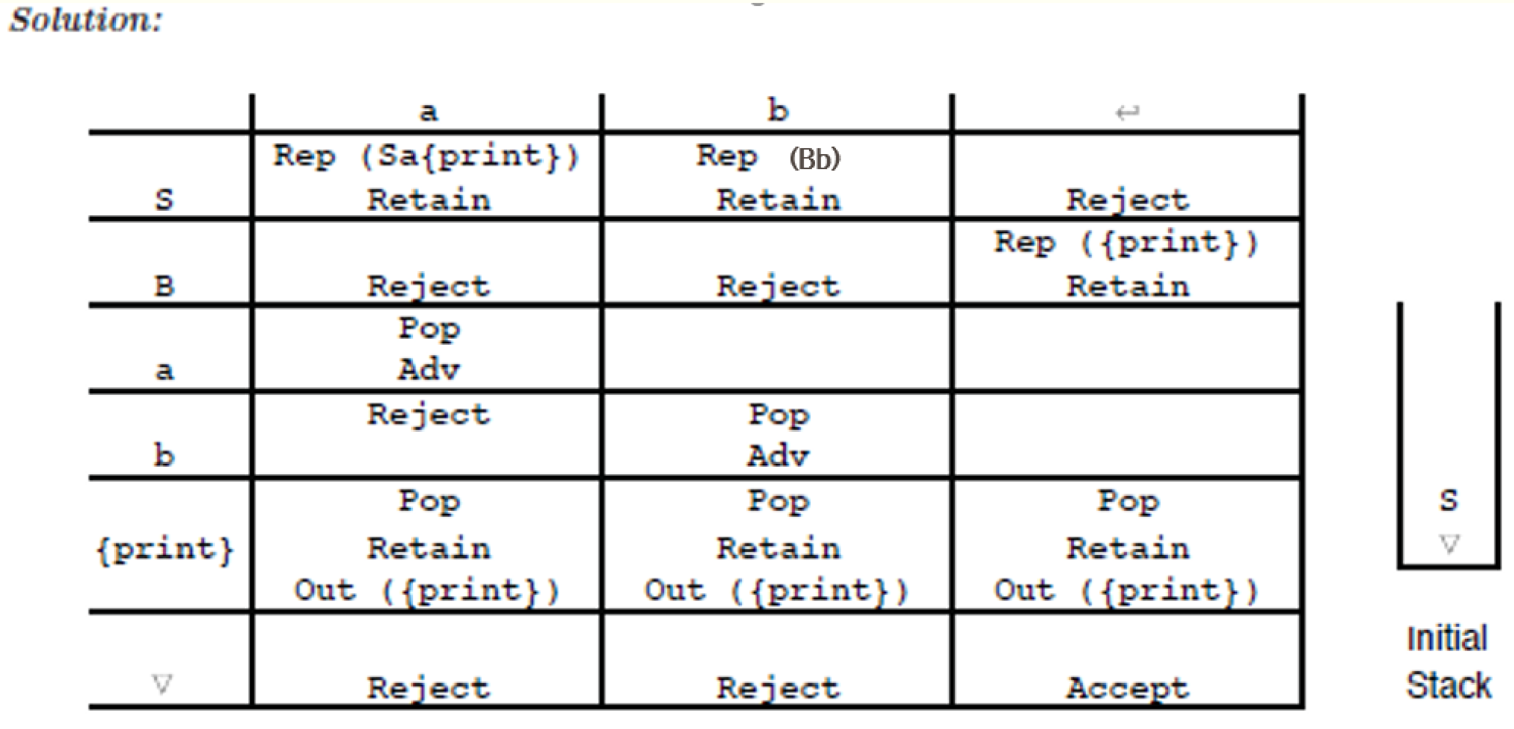
Exercise

- Show an extended pushdown translator for the translation grammar G18.



G18:

1. S → {print}aS
2. S → bB
3. B → {print}



Solution:

	a	b	↩
S	Rep (Sa{print}) Retain	Rep (Bb) 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

END OF SYNTAX-DIRECTED TRANSLATION

Next: Chapter 5

ATTRIBUTED GRAMMARS

Introduction

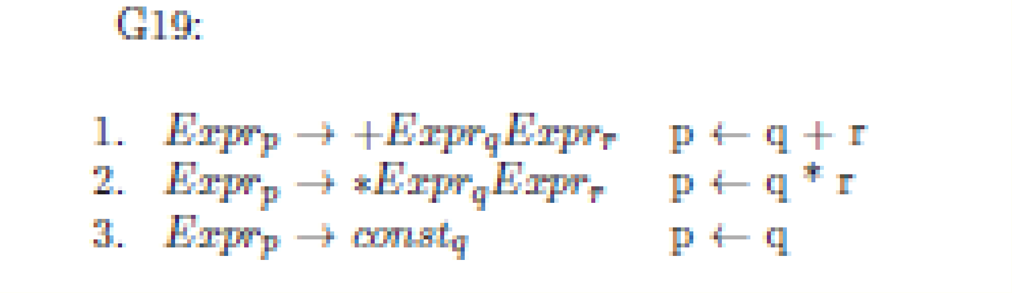
An attribute grammar is a context-free grammar with the addition of attributes and attribute evaluation rules called semantic functions.

Thus, an attribute grammar can specify both semantics and syntax while BNF specifies only the syntax

each of the terminals and nonterminals may have zero or more attributes, normally designated by subscripts, associated with it.

There are two types: Syntesized attributes and inherited attributes

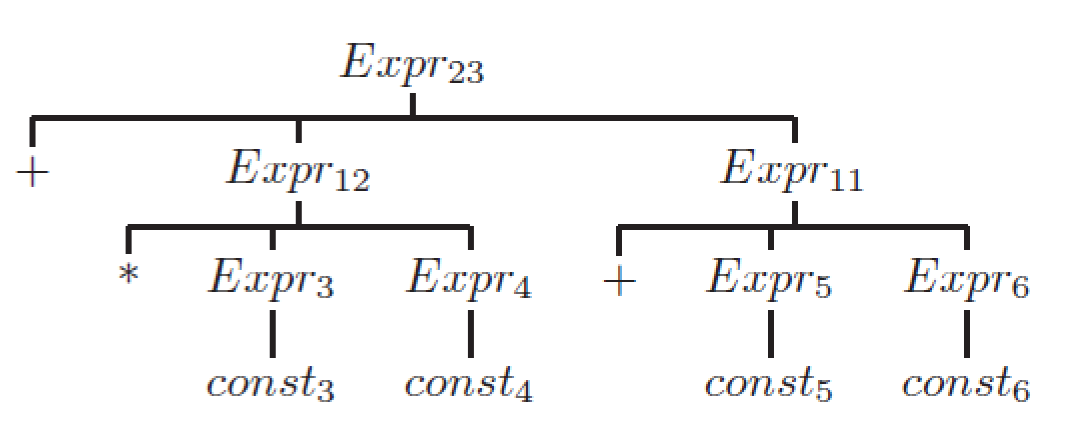
Example: Synthesized Attributes



G19:

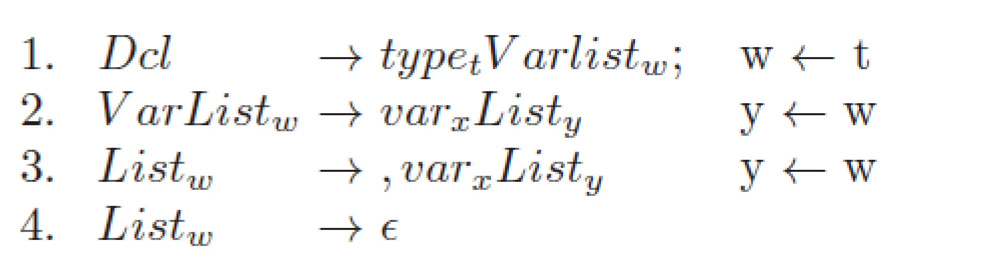
1. $Expr_{p} \rightarrow +Expr_{q}Expr_{r}$ p ← q + r
2. $Expr_{p} \rightarrow *Expr_{q}Expr_{r}$ p ← q * r
3. $Expr_{p} \rightarrow const_{q}$ p ← q

An attributed derivation tree for the prefix expression + * 3 4 + 5 6 using grammar G19



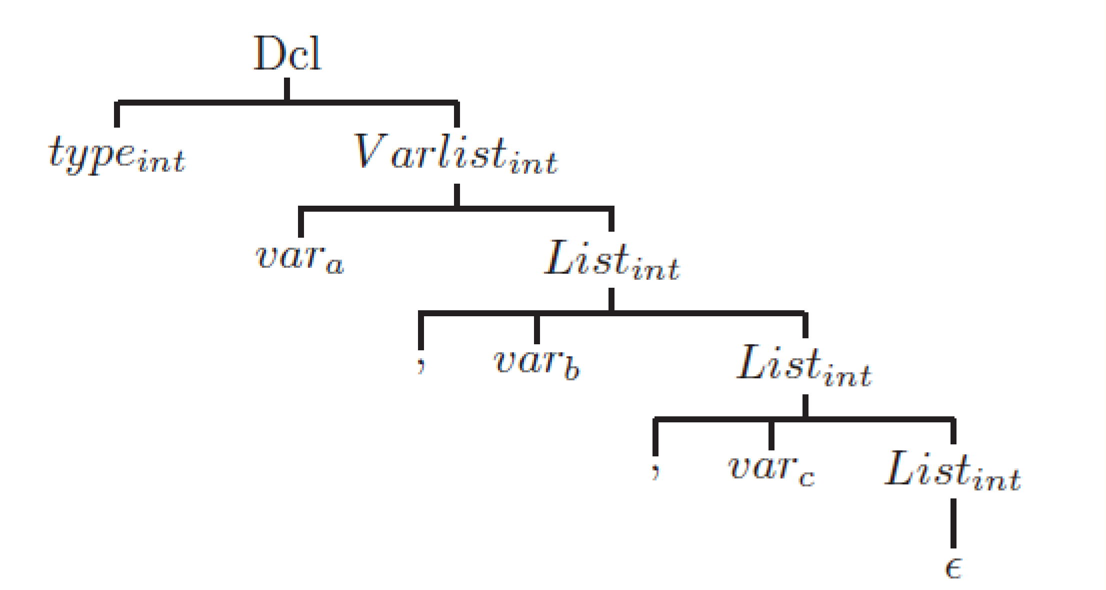
Example: Inherited attributes

- Grammar G20



1.	Dcl	$\rightarrow type_t Varlist_w;$	$\mathrm{w} \leftarrow \mathrm{t}$
2.	$VarList_w$	$\rightarrow var_x List_y$	$\mathrm{y} \leftarrow \mathrm{w}$
3.	$List_w$	$\rightarrow , var_x List_y$	$\mathrm{y} \leftarrow \mathrm{w}$
4.	$List_w$	$\rightarrow \epsilon$	

An attributed derivation tree for int a,b,c; using grammar G20



Implementing Attributed Grammars with Recursive Descent

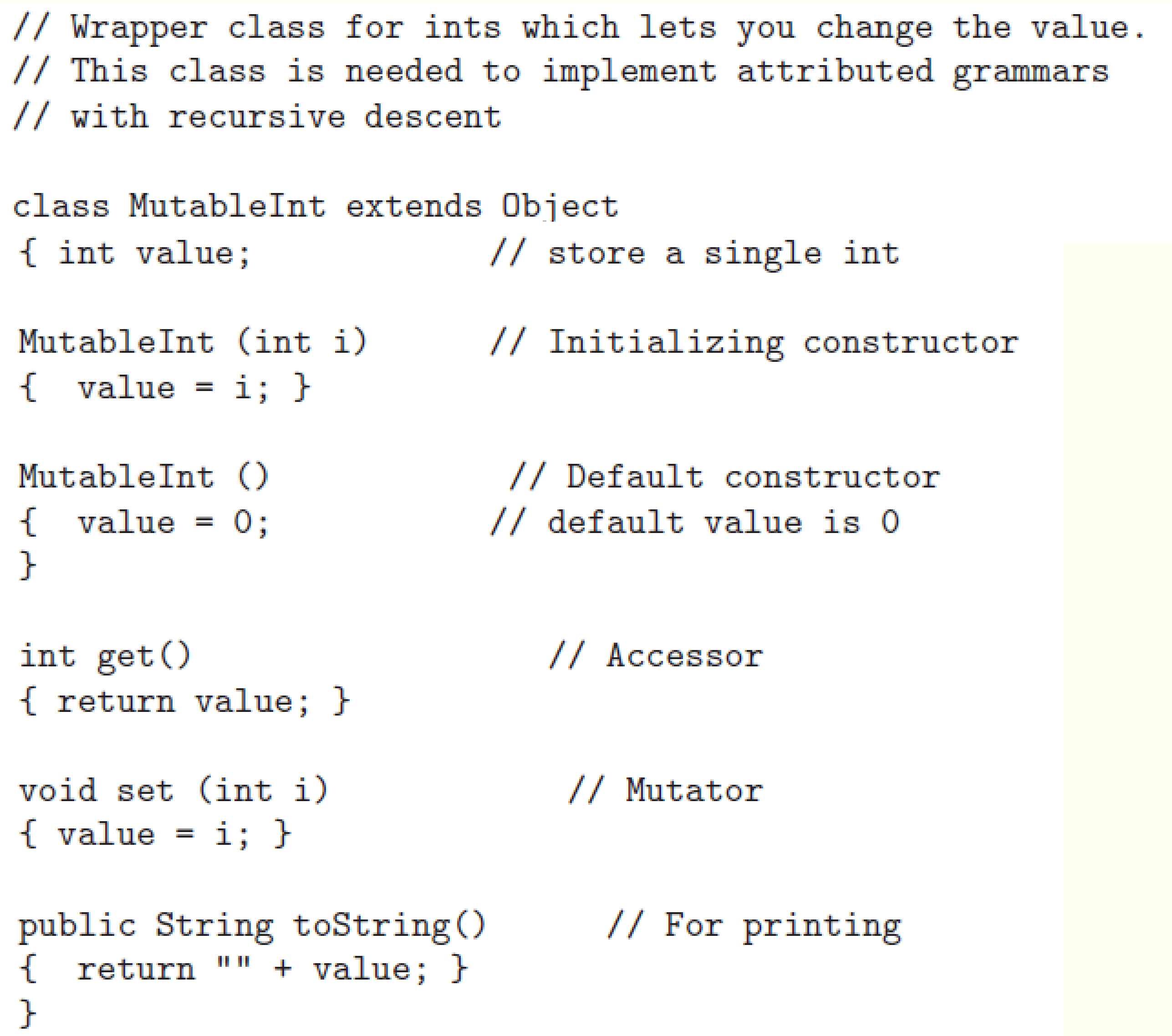
- To implement an attributed grammar with recursive descent, the attributes will be implemented as parameters or instance variables in the methods defining nonterminals.

- For example, if Sa,b is a nonterminal with two attributes, then the method S will have two parameters, a and b.
    - Methos S (datatype s, datatype b)

- Synthesized attributes are used to return information to the calling method and, hence, must be implemented with objects (i.e. with reference types)

- Notes
    - Synthesized attributes is used to return information to calling method)
    - Inherited attributes is used to pass information to called method.

Example: Attribute to store whole number

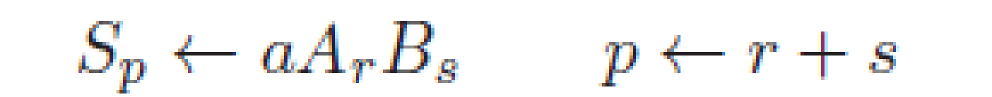


```
// Wrapper class for ints which lets you change the value.
// This class is needed to implement attributed grammars
// with recursive descent

class MutableInt extends Object
{ int value;            // store a single int

MutableInt (int i)      // Initializing constructor
{  value = i; }

MutableInt ()            // Default constructor
{  value = 0;           // default value is 0
}

int get()                  // Accessor
{ return value; }

void set (int i)            // Mutator
{ value = i; }

public String toString()      // For printing
{  return "" + value; }
}
```

Revision:
Creating object in JAVA

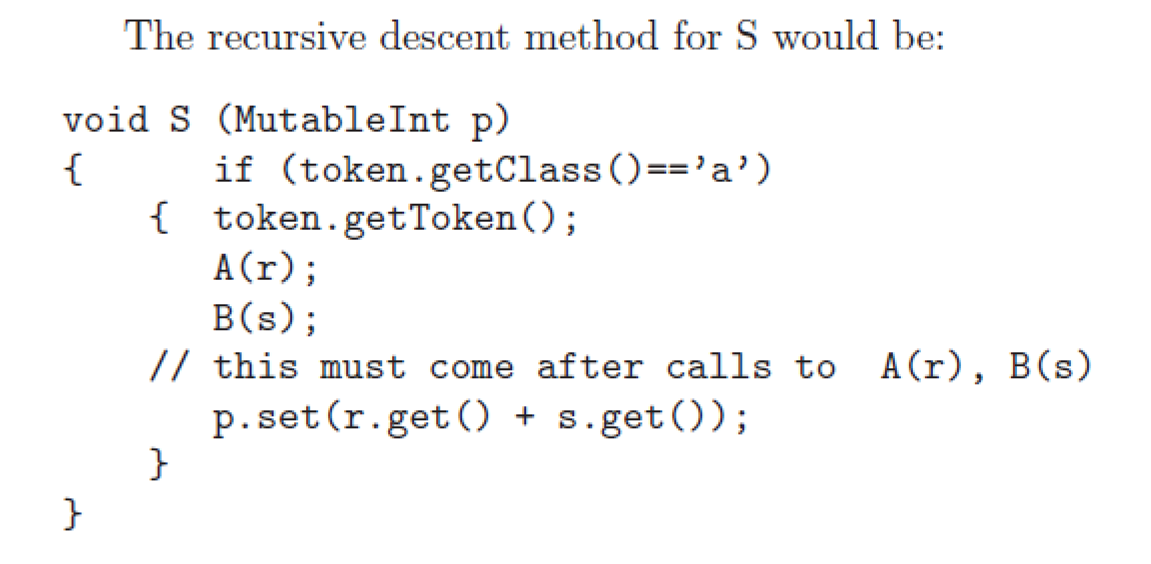
1. Create class
2. Declare variable
3. Create constructor
4. Create get method
5. Create set method
6. Print method

Example : Recursive descent



$$S_p \leftarrow a A_r B_s \qquad p \leftarrow r + s$$

Action symbol attributes will be implemented as instance variables



The recursive descent method for S would be:

```
void S (MutableInt p)
{      if (token.getClass()==’a’)
    {  token.getToken();
       A(r);
       B(s);
    // this must come after calls to  A(r), B(s)
       p.set(r.get() + s.get());
    }
}
```

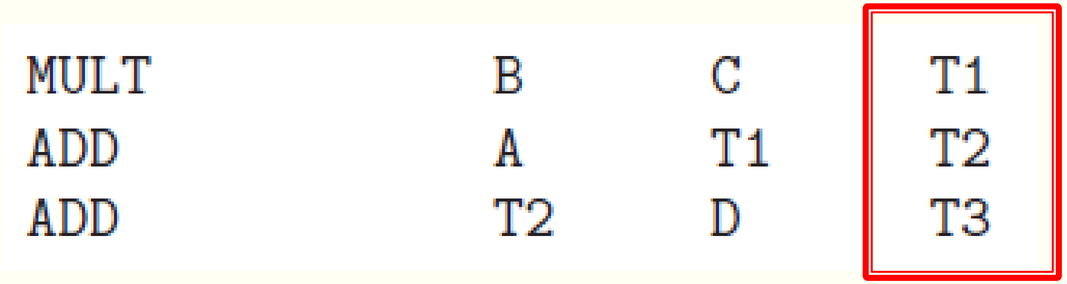
AN ATTRIBUTED TRANSLATION GRAMMAR FOR EXPRESSIONS

Introduction

- The output of the translator will be a stream of atoms.

- Each atom will consist of four parts:
  - (1) an operation, ADD or MULT,
  - (2) a left operand,
  - (3) a right operand, and
  - (4) a result.

- For example: A + B * C + D



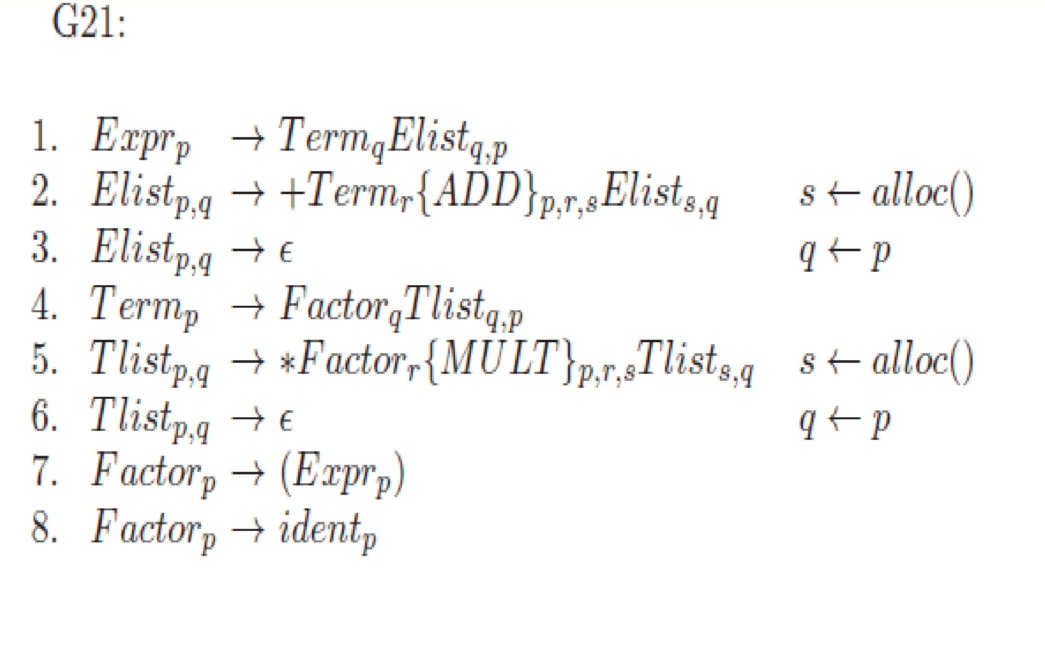
MULT	B	C	T1
ADD	A	T1	T2
ADD	T2	D	T3

- Translator will have to find **temporary storage** locations (or use a stack) to store intermediate results at run time.

- The final result of the expression is in T3

The attributed translation grammar G21

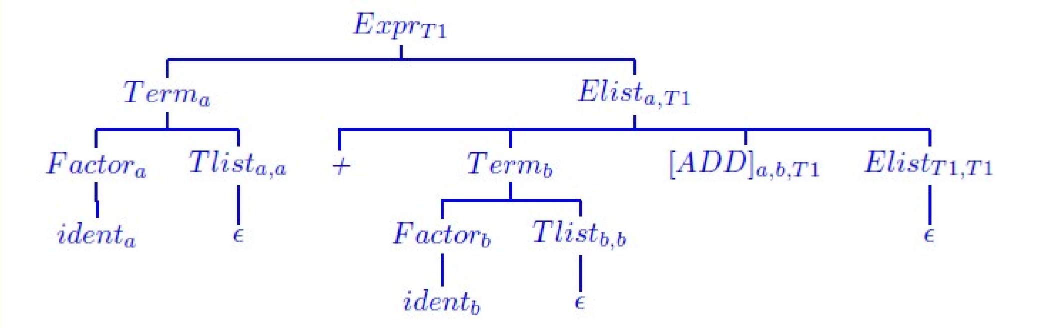
- all nonterminal attributes are synthesized, with the exception of the first attribute on Elist and Tlist, which are inherited



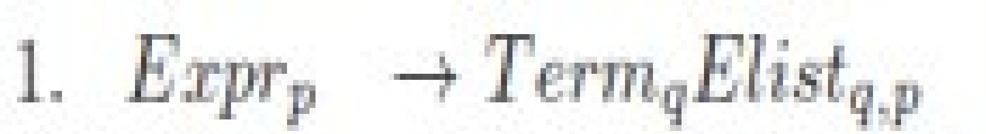
G21:

1. $Expr_p \rightarrow Term_q Elist_{q,p}$
2. $Elist_{p,q} \rightarrow +Term_r \{ADD\}_{p,r,s} Elist_{s,q} \qquad s \leftarrow alloc()$
3. $Elist_{p,q} \rightarrow \epsilon \qquad q \leftarrow p$
4. $Term_p \rightarrow Factor_q Tlist_{q,p}$
5. $Tlist_{p,q} \rightarrow *Factor_r \{MULT\}_{p,r,s} Tlist_{s,q} \qquad s \leftarrow alloc()$
6. $Tlist_{p,q} \rightarrow \epsilon \qquad q \leftarrow p$
7. $Factor_p \rightarrow (Expr_p)$
8. $Factor_p \rightarrow ident_p$

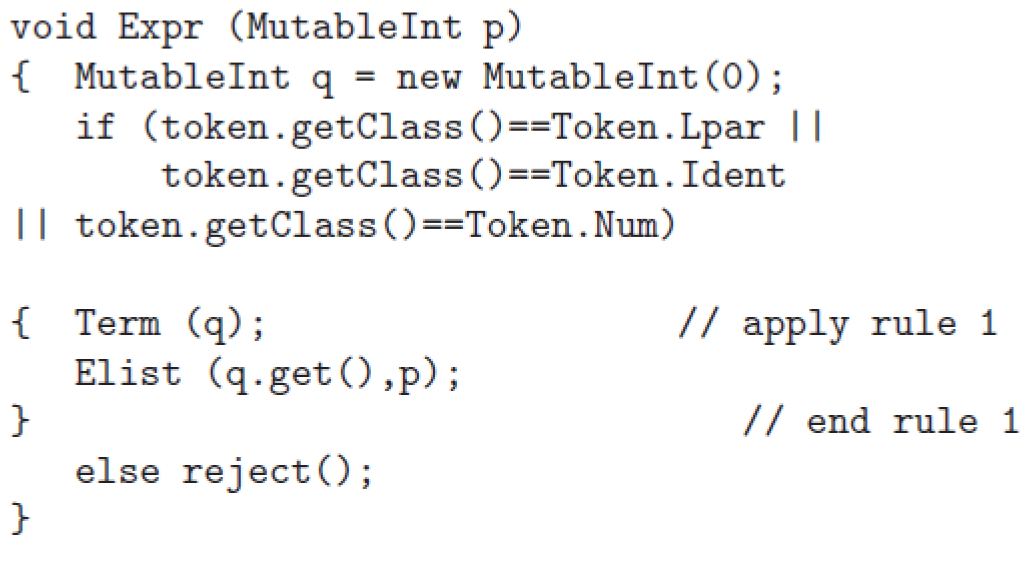
- Show an attributed derivation tree for the expression a+b using grammar G21.



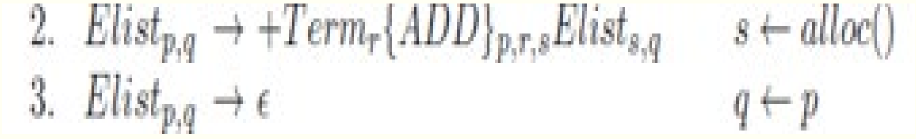
Translating Expressions with Recursive Descent



1. $Expr_p \rightarrow Term_q Elist_{q,p}$

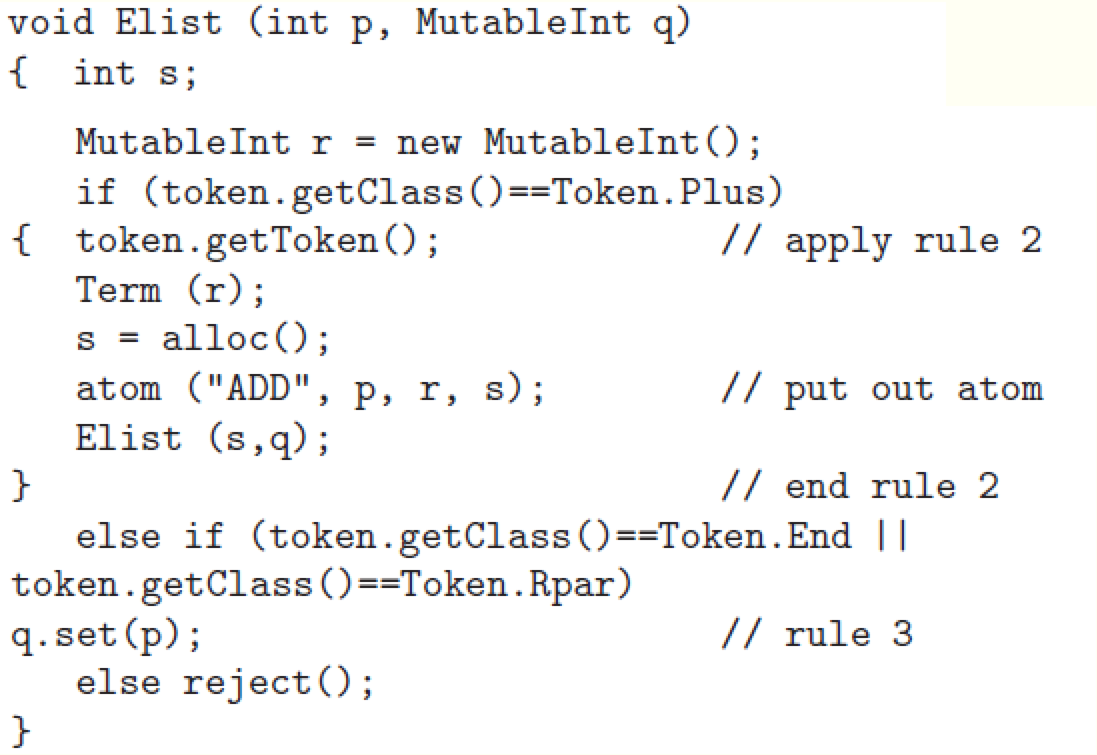


```
void Expr (MutableInt p)
{  MutableInt q = new MutableInt(0);
   if (token.getClass()==Token.Lpar ||
       token.getClass()==Token.Ident
|| token.getClass()==Token.Num)

{  Term (q);                      // apply rule 1
   Elist (q.get(),p);
}                                    // end rule 1
   else reject();
}
```



2. $Elist_{p,q} \rightarrow +Term_r\{ADD\}_{p,r,s}Elist_{s,q}$ $\quad s \leftarrow alloc()$

3. $Elist_{p,q} \rightarrow \epsilon$ $\quad q \leftarrow p$

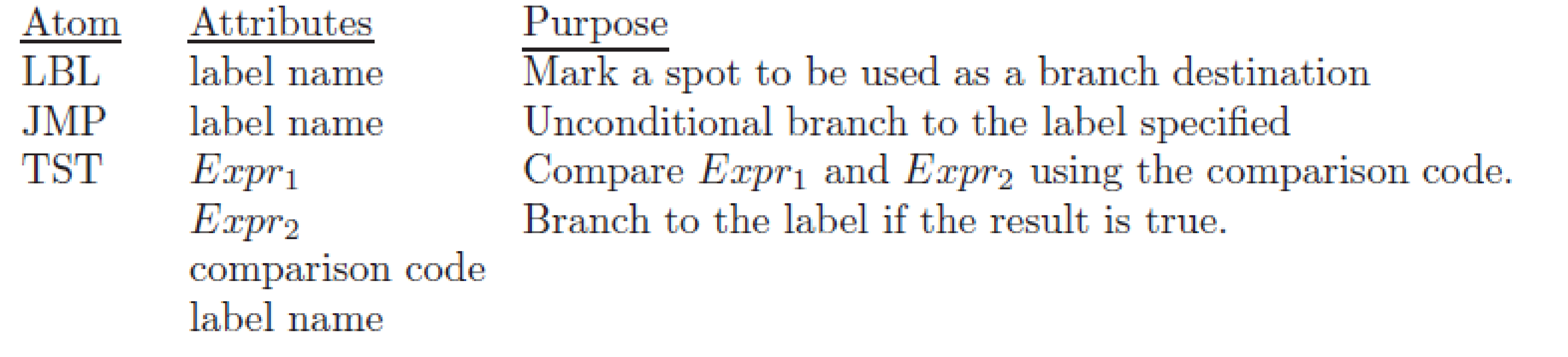


```
void Elist (int p, MutableInt q)
{  int s;
   MutableInt r = new MutableInt();
   if (token.getClass()==Token.Plus)
{  token.getToken();             // apply rule 2
   Term (r);
   s = alloc();
   atom ("ADD", p, r, s);        // put out atom
   Elist (s,q);
}                                // end rule 2
   else if (token.getClass()==Token.End ||
token.getClass()==Token.Rpar)
q.set(p);                        // rule 3
   else reject();
}
```

EXPRESSIONS FOR A SUBSET OF A LANGUAGE

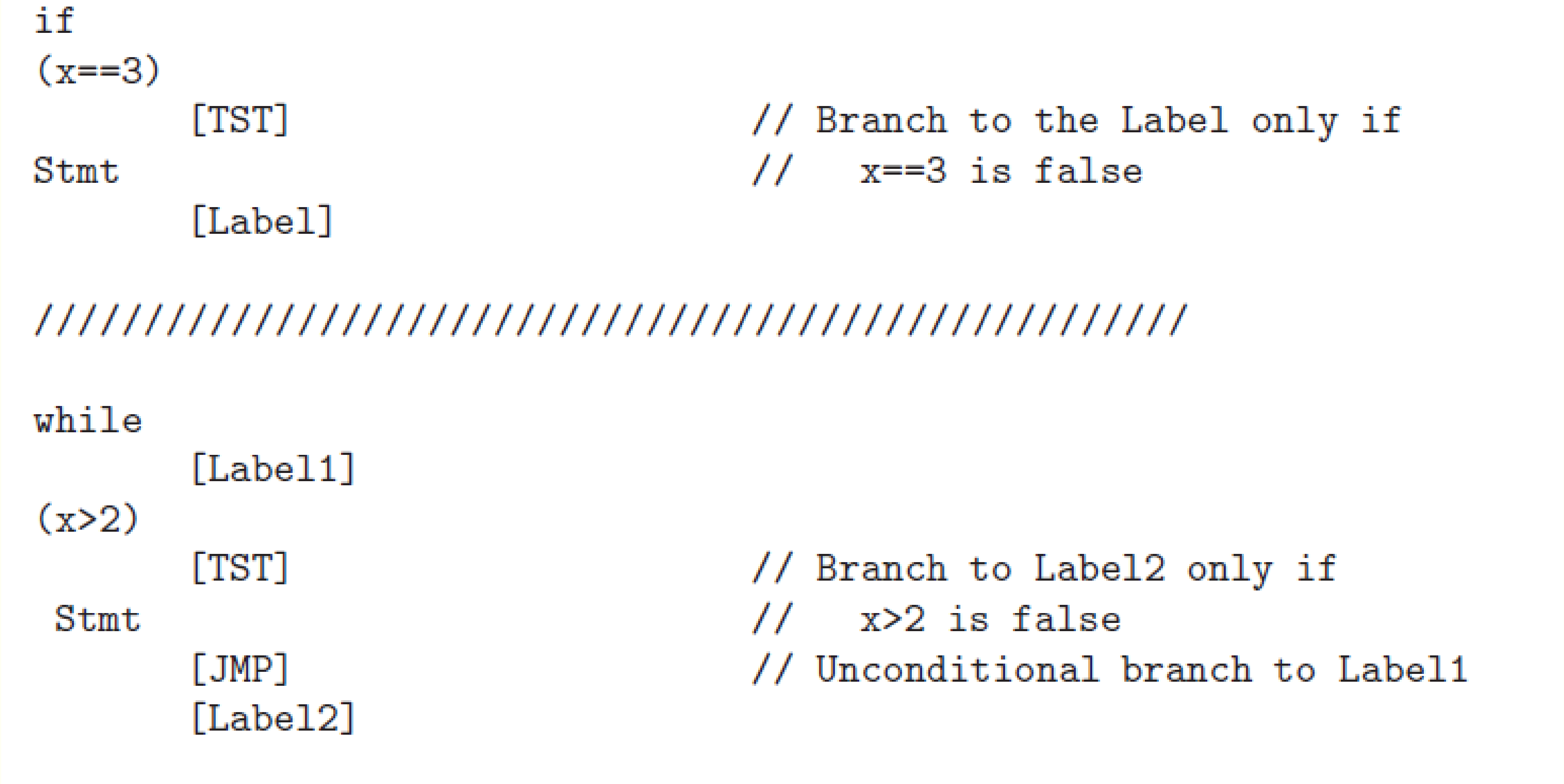
Next: Chapter 5

LBL, JMP, TST, and MOV atoms



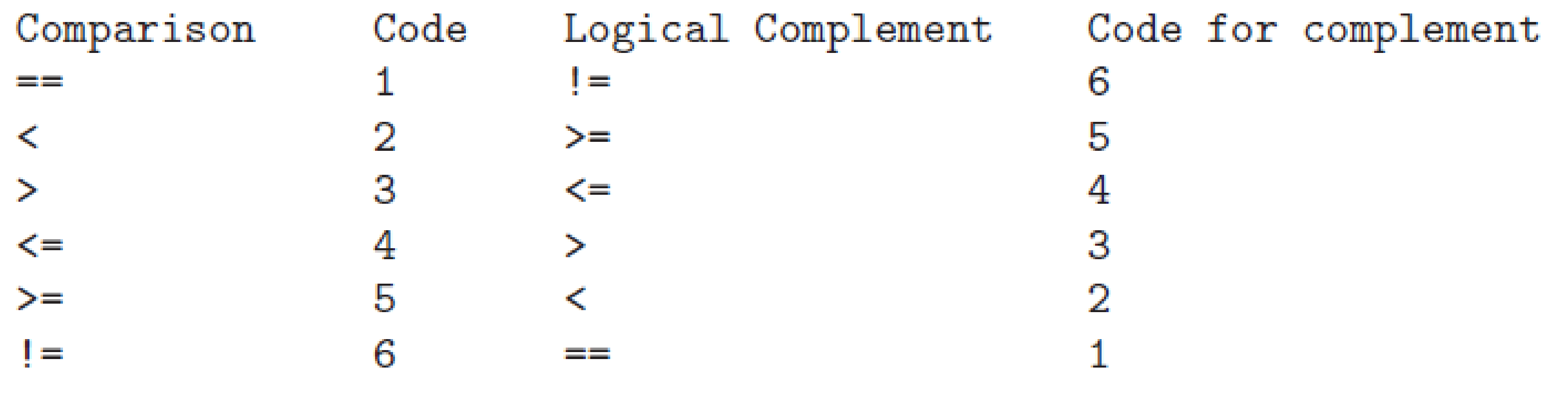
Atom	Attributes	Purpose
LBL	label name	Mark a spot to be used as a branch destination
JMP	label name	Unconditional branch to the label specified
TST	$Expr_1$ $Expr_2$ comparison code label name	Compare $Expr_1$ and $Expr_2$ using the comparison code. Branch to the label if the result is true.

Example



```
if
(x==3)
       [TST]                         // Branch to the Label only if
Stmt                                 //   x==3 is false
       [Label]

/////////////////////////////////////////////////////


while
       [Label1]
(x>2)
       [TST]                         // Branch to Label2 only if
 Stmt                                //   x>2 is false
       [JMP]                         // Unconditional branch to Label1
       [Label2]
```

Boolean expressions



Comparison	Code	Logical Complement	Code for complement
==	1	!=	6
<	2	>=	5
>	3	<=	4
<=	4	>	3
>=	5	<	2
!=	6	==	1

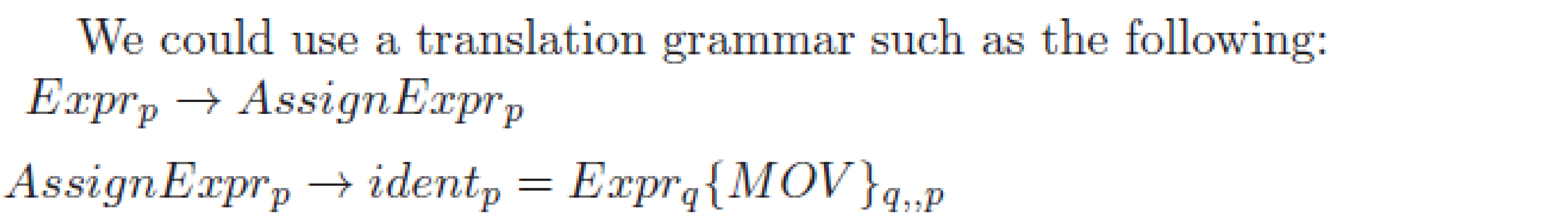
To process a boolean expression all we need to do is put out a TST atom which allocates a new label name and branches to that label when the comparison is false.

$$BoolExpr_{Lbl} \rightarrow Expr_{p} \; compare_{c} \; Expr_{q} \; \{TST\}_{p,q,,7-c,Lbl}$$

will compare the values stored at a and b, using the comparison whose code is c, and branch to a label designated x if the comparison is true.

Assignment

- Assignment statement always used to assign the value of the right operand to the variable which is the left operand.
- Therefore, it needs a MOV atom to implement the assignment.

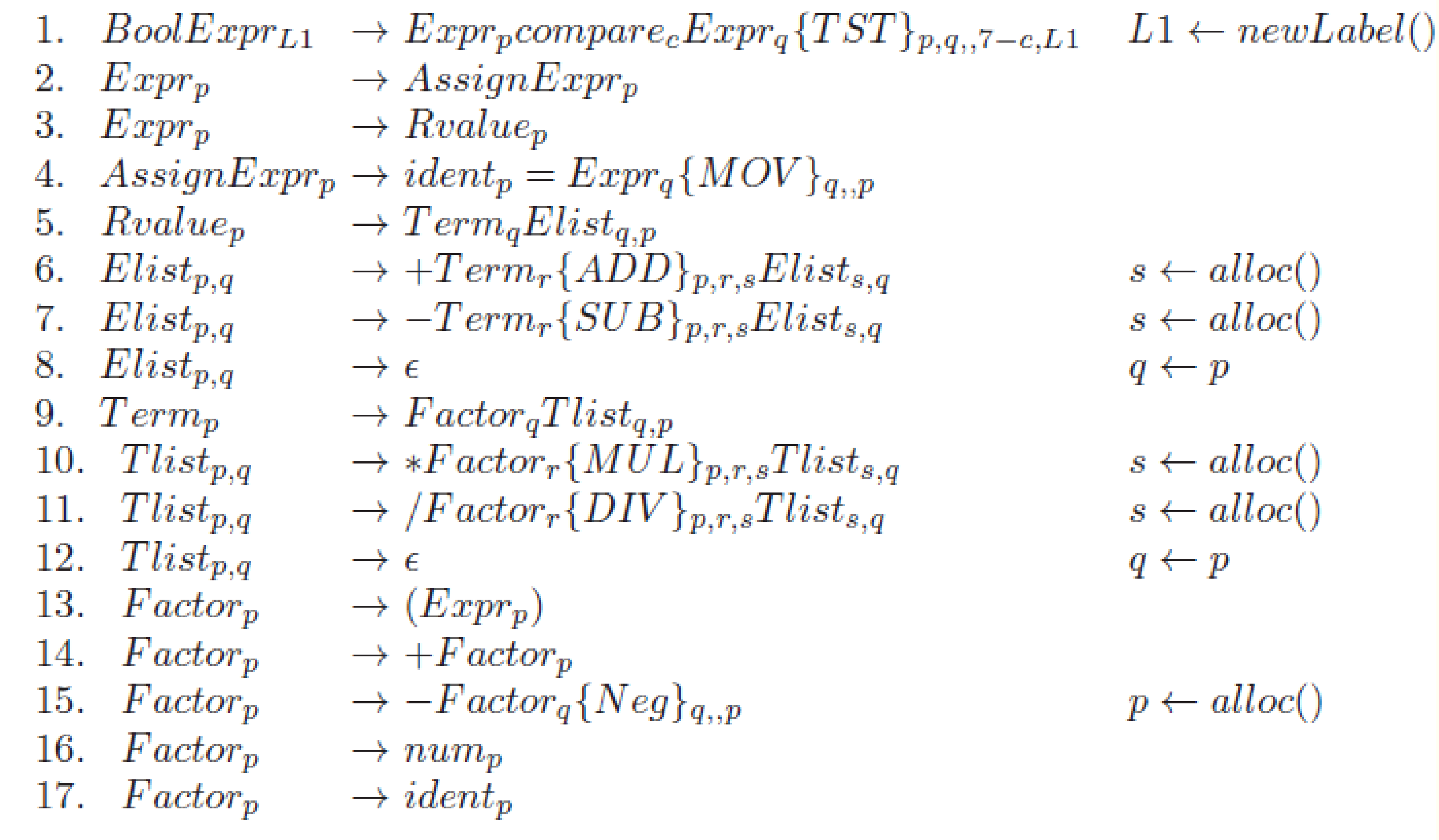


We could use a translation grammar such as the following:

$$Expr_p \rightarrow AssignExpr_p$$

$$AssignExpr_p \rightarrow ident_p = Expr_q\{MOV\}_{q,,p}$$

Attribute Translation grammar for Decaf (Add, Sub, Mul, Div)



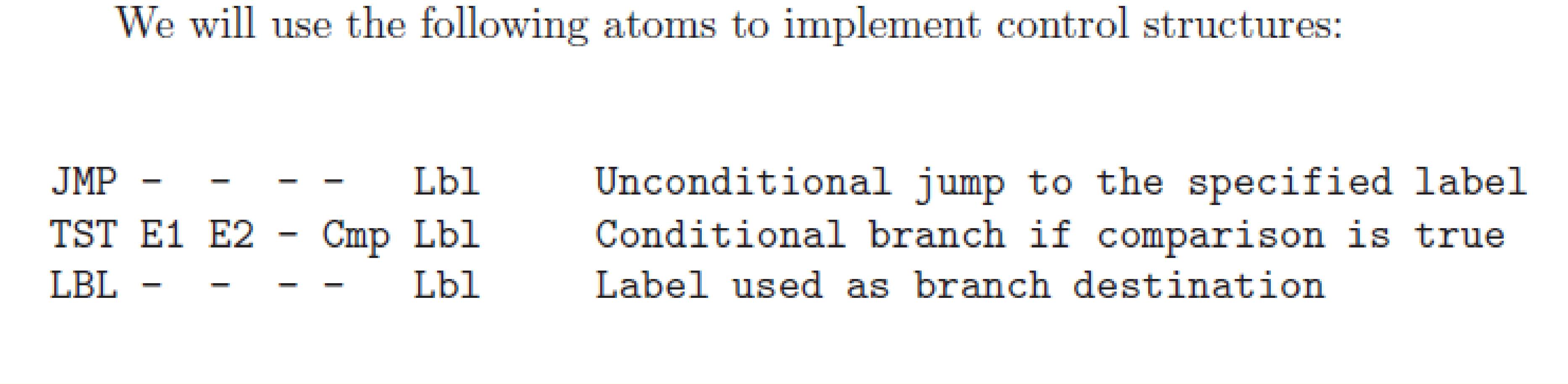
1.	$BoolExpr_{L1}$	$\rightarrow Expr_p compare_c Expr_q \{TST\}_{p,q,,7-c,L1}$	$L1 \leftarrow newLabel()$
2.	$Expr_p$	$\rightarrow AssignExpr_p$	
3.	$Expr_p$	$\rightarrow Rvalue_p$	
4.	$AssignExpr_p$	$\rightarrow ident_p = Expr_q \{MOV\}_{q,,p}$	
5.	$Rvalue_p$	$\rightarrow Term_q Elist_{q,p}$	
6.	$Elist_{p,q}$	$\rightarrow +Term_r \{ADD\}_{p,r,s} Elist_{s,q}$	$s \leftarrow alloc()$
7.	$Elist_{p,q}$	$\rightarrow -Term_r \{SUB\}_{p,r,s} Elist_{s,q}$	$s \leftarrow alloc()$
8.	$Elist_{p,q}$	$\rightarrow \epsilon$	$q \leftarrow p$
9.	$Term_p$	$\rightarrow Factor_q Tlist_{q,p}$	
10.	$Tlist_{p,q}$	$\rightarrow *Factor_r \{MUL\}_{p,r,s} Tlist_{s,q}$	$s \leftarrow alloc()$
11.	$Tlist_{p,q}$	$\rightarrow /Factor_r \{DIV\}_{p,r,s} Tlist_{s,q}$	$s \leftarrow alloc()$
12.	$Tlist_{p,q}$	$\rightarrow \epsilon$	$q \leftarrow p$
13.	$Factor_p$	$\rightarrow (Expr_p)$	
14.	$Factor_p$	$\rightarrow +Factor_p$	
15.	$Factor_p$	$\rightarrow -Factor_q \{Neg\}_{q,,p}$	$p \leftarrow alloc()$
16.	$Factor_p$	$\rightarrow num_p$	
17.	$Factor_p$	$\rightarrow ident_p$	

TRANSLATING CONTROL STRUCTURE

Next: Chapter 5

- In order to translate control structures, such as for, while, and if statements, we must first consider simple operations such as Unconditional Jump or Goto, Compare, and Conditional Jump.

- In order to implement these Jump operations, we need to establish a jump, or destination address.



We will use the following atoms to implement control structures:

```
JMP -  -  - -   Lbl     Unconditional jump to the specified label
TST E1 E2 - Cmp Lbl     Conditional branch if comparison is true
LBL -  -  - -   Lbl     Label used as branch destination
```

The purpose of the TST atom is to compare the values of expressions E1 and E2 using the specified comparison operator, Cmp, and then branch to the label Lbl if the comparison is true.

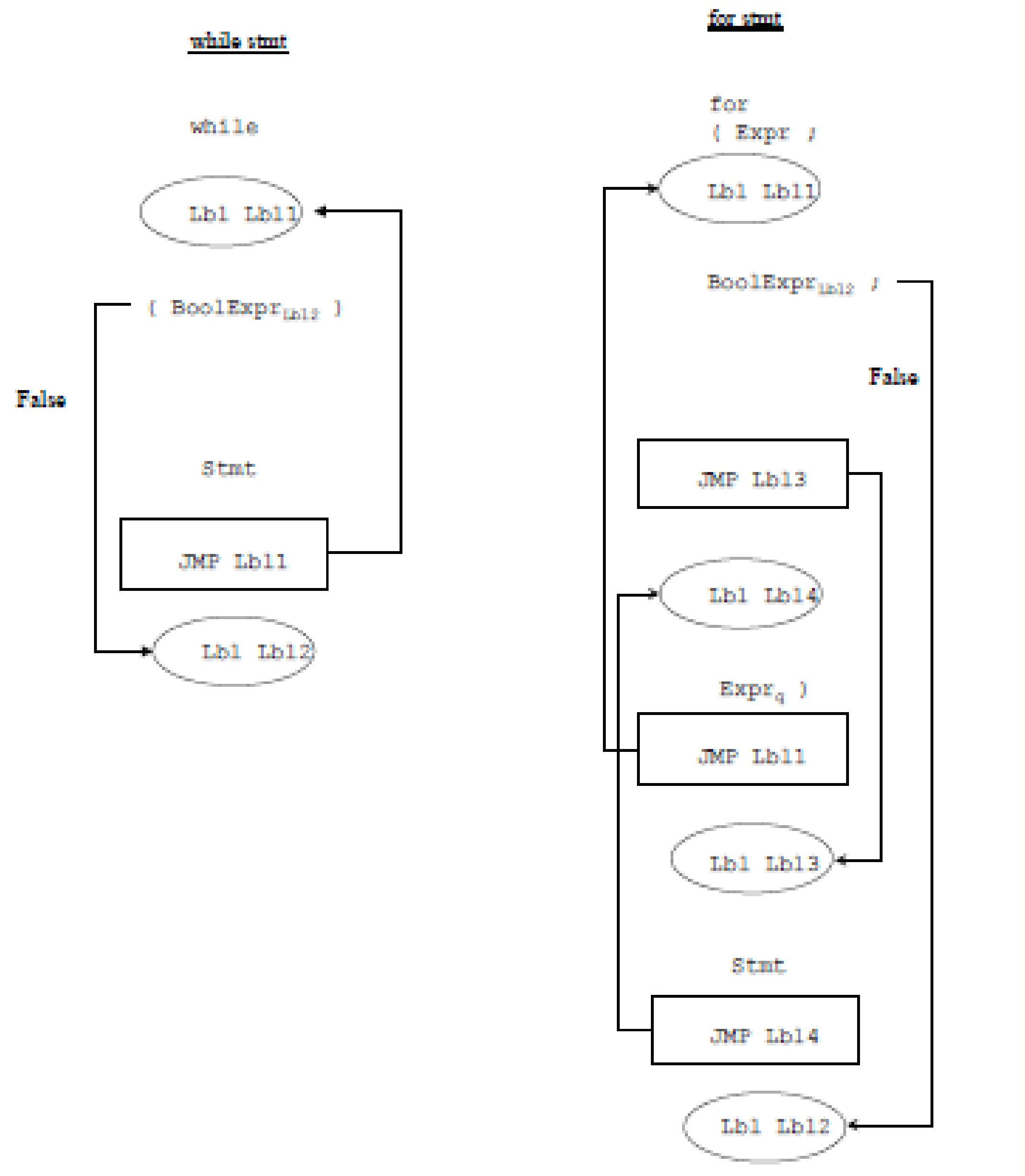
Comparison operator

==	is 1	<=	is 4
<	is 2	>=	is 5
>	is 3	!=	is 6

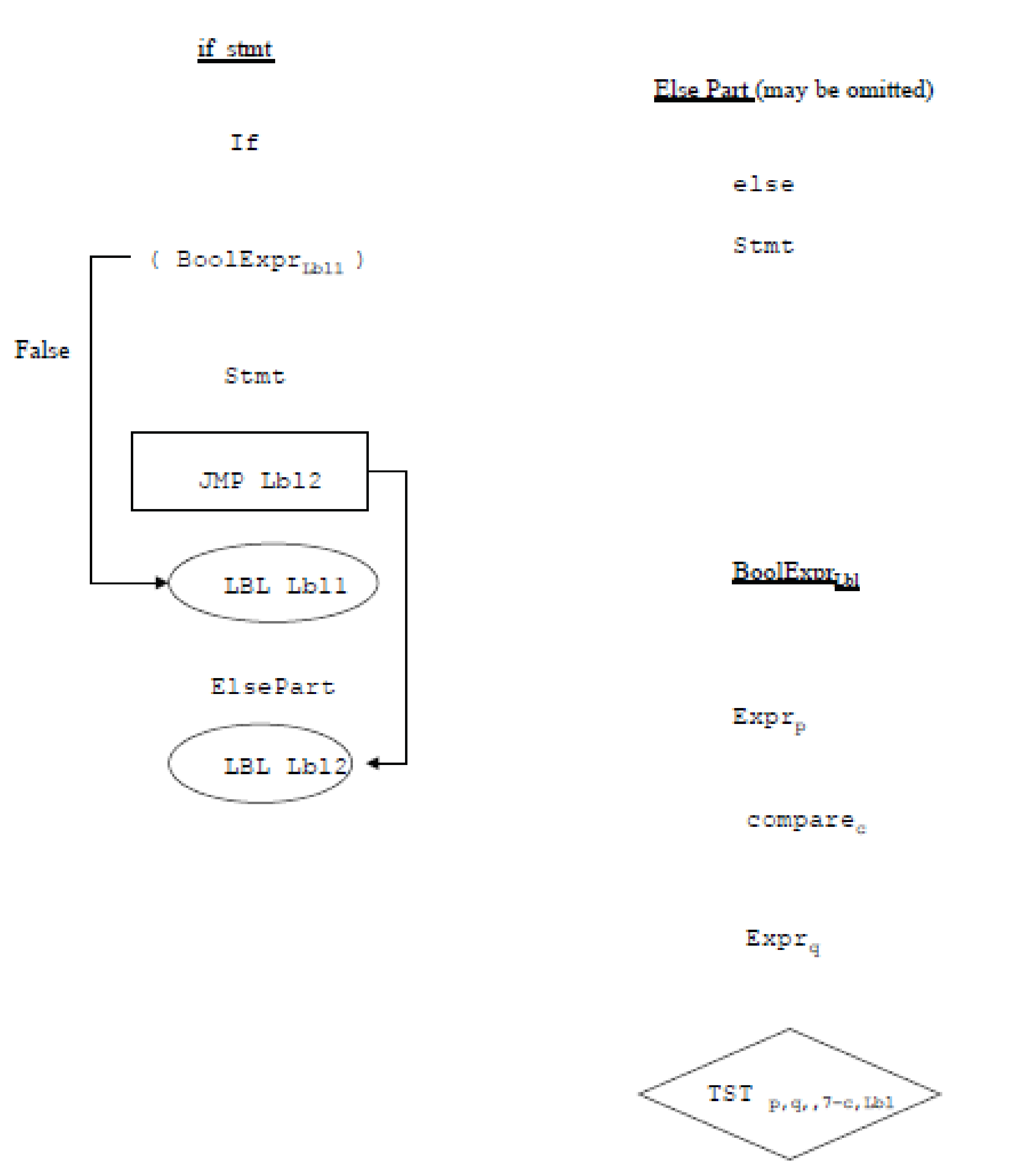
For example: **TST,A,C, ,4,L3**

means jump to label L3 if A is less than or equal to C.

1. $Stmt \rightarrow while(BoolExpr)Stmt$
2. $Stmt \rightarrow for(Expr; BoolExpr; Expr)Stmt$



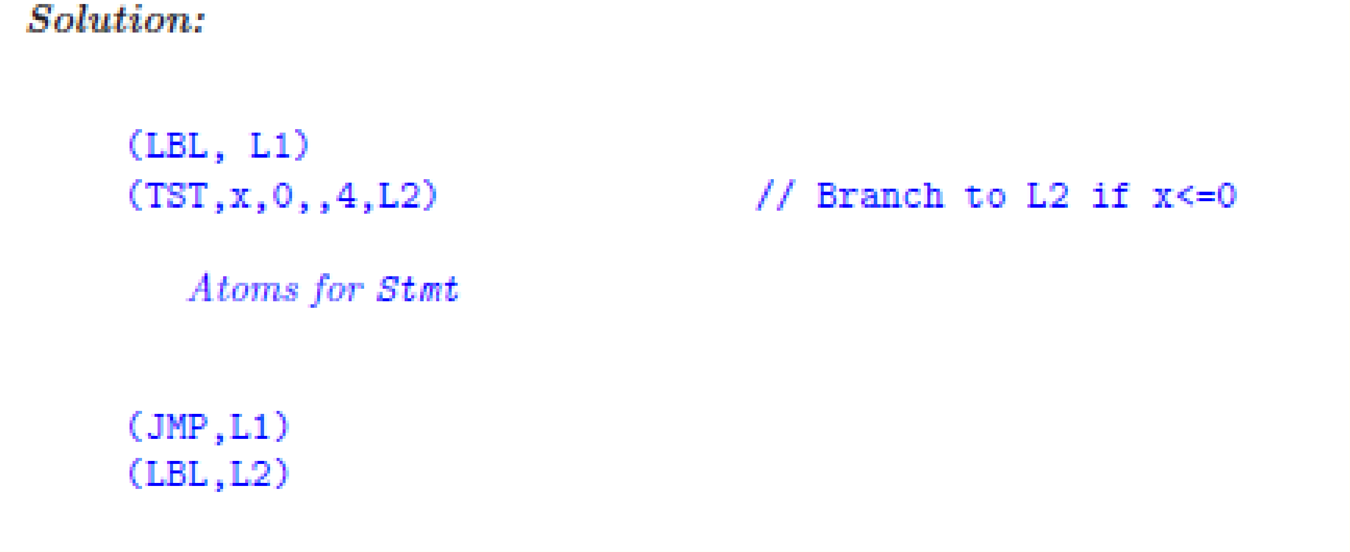
$$Stmt \rightarrow if(BoolExpr)Stmt\ ElsePart$$
$$ElsePart \rightarrow else\ Stmt$$
$$ElsePart \rightarrow \epsilon$$



Exercise

Show the atom string which would be put out that corresponds to the following Java statement:

while (x > 0) Stmt



Solution:

```
(LBL, L1)
(TST,x,0,,4,L2)                 // Branch to L2 if x<=0

   Atoms for Stmt


(JMP,L1)
(LBL,L2)
```

END OF ATTRIBUTED GRAMMARS

Next: Chapter 5