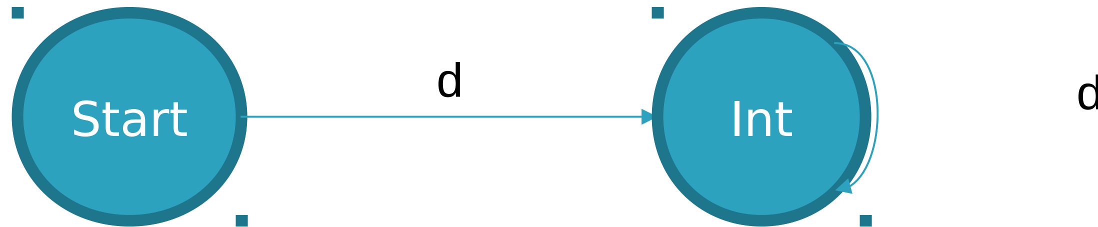
Chapter-3
Scanning and Parsing

# Scanning

- Scanning is process of recognizing the lexical components in a source string.
- Lexical features of a language can be specified using Type-3 or regular grammars.
- Reasons for separating scanning from parsing.
  - A recognizer Type-3 productions is simpler, easier to build and more efficient during execution than a recognizer for type-2 productions.

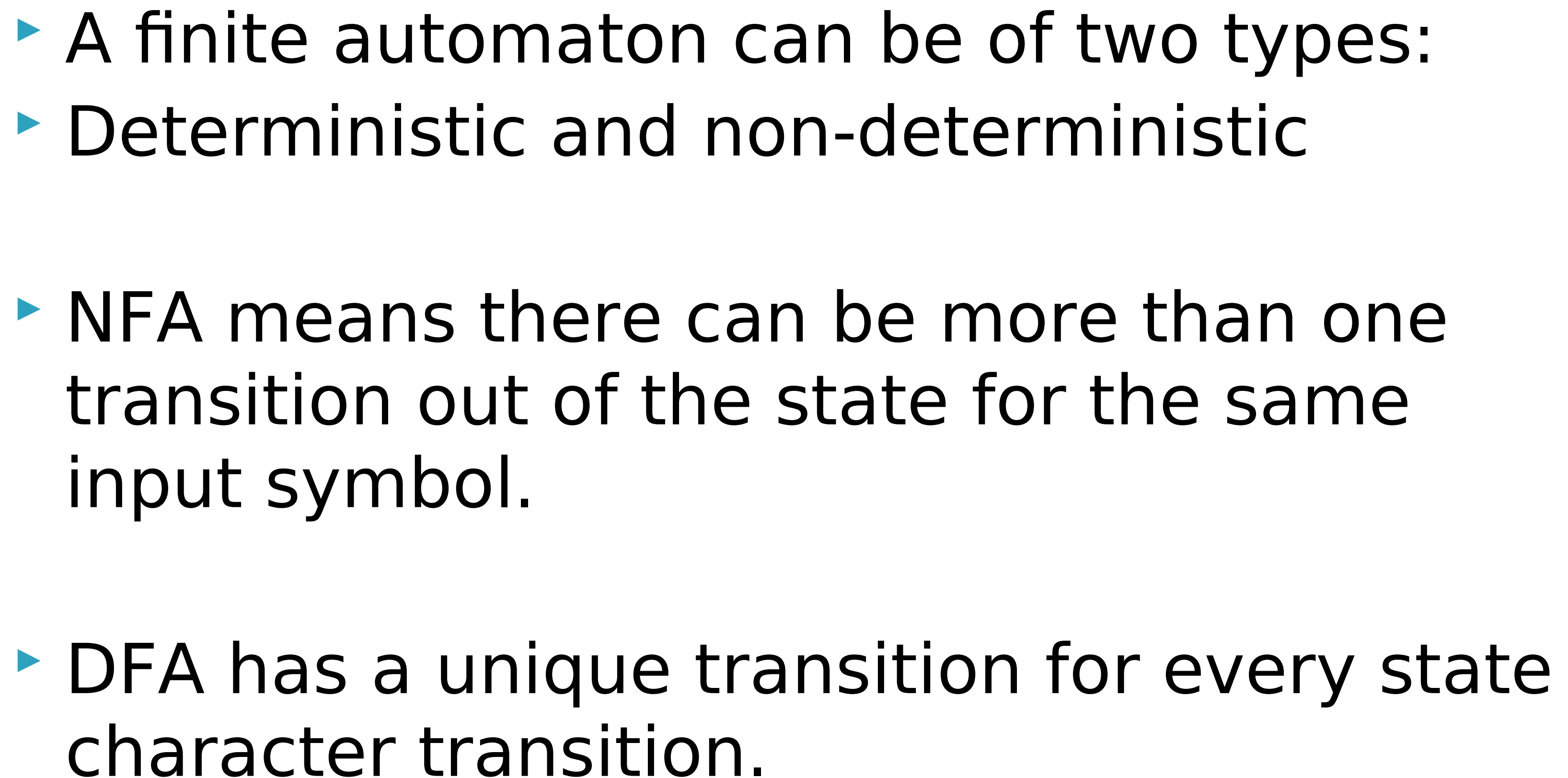
# Transition Diagram

- During the construction of a lexical analyser, convert the pattern in the diagrammatic form called transition diagram.
- Transition diagrams are used to keep track of recognized sequence of characters as the forward pointer scans the input.



# Finite state automata

- Finite automata are graphs that decide whether a sentence is in the language (set of valid string generated by regular expression) or not.
- A finite state automaton (FSA) is triples (S, Σ, T) where
  - S : is a finite set of states, one of which is the initial state $S_{init,}$ and one or more of which are the final states.
  - Σ : is the alphabet of source symbols.
  - T : is finite set of state transitions defining transitions out of each si ε S in encountering the symbols of Σ.



- A finite automaton can be of two types:
- Deterministic and non-deterministic

- NFA means there can be more than one transition out of the state for the same input symbol.

- DFA has a unique transition for every state character transition.

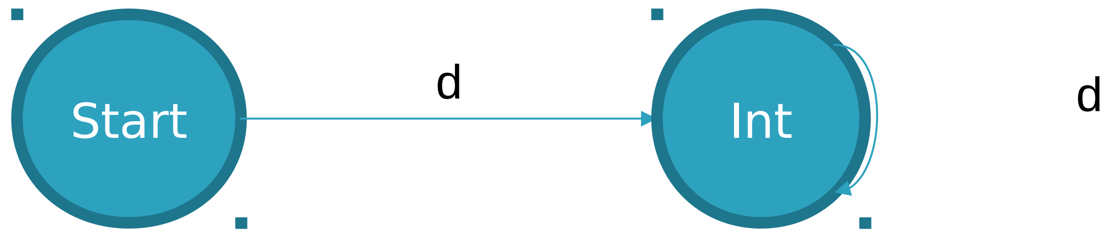
# DFA

- Deterministic finite state automaton (DFA) is an FSA such that t1ε T, t1=(si,symb,sj) implies not exist t2 ε T, t2=(si, symb, sk). No state has $\mathcal{E}$ transition.
- At most one transition exists in state si for symbol symb.
- The DFA halts when all symbols in the source string are recognized, or error condition is encountered.

# Cont.

- Example :

  <integer>=d|<integer>d




| State | Next Symbol |
|---|---|
| start | Int |
| Int | Int |

# Regular expression

- The generalization of type-3 production called a regular expression.
- A regular expression matches a set of strings
- Example:

  An organization uses an employee code which contain section code and numeric code.

  <section code>=l|<section code>l

  <numeric code>= d|<numeric code> d

  <employee code> = <section code> <numeric code>

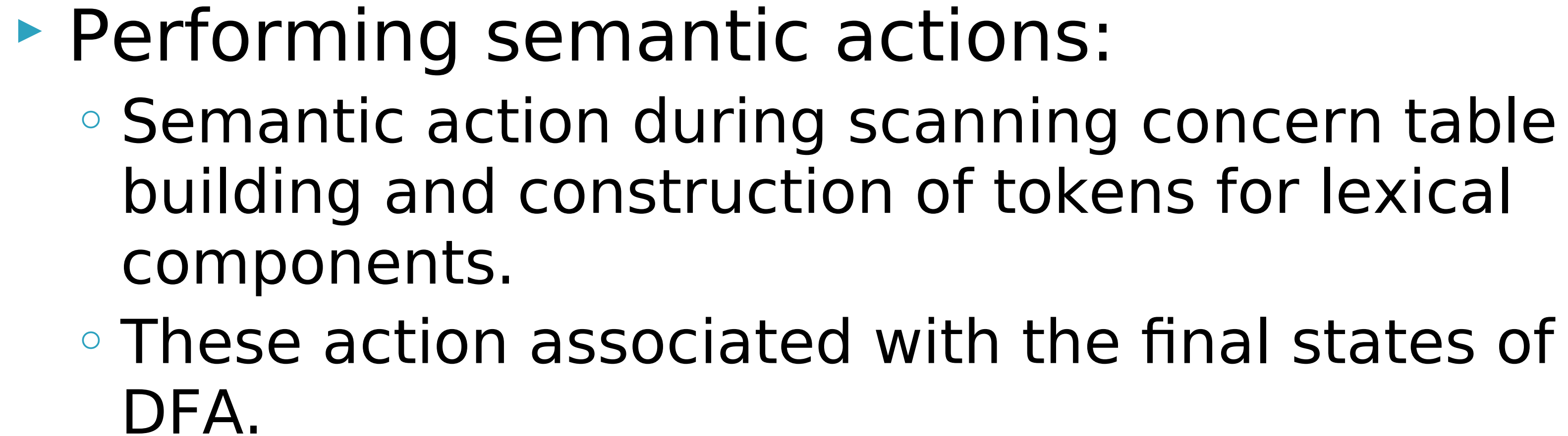
Regular expression for employee code is **(l)**$^{+}$ **(d)**$^{+}$ .

# Cont.

- Regular expression

<table>
<tr><th>Regular expression</th><th>Meaning</th></tr>
<tr><td>r</td><td>String r</td></tr>
<tr><td>s</td><td>String s</td></tr>
<tr><td>r.s or rs</td><td>Concatenation of r and s</td></tr>
<tr><td>(r)</td><td>same meaning as r</td></tr>
<tr><td>r|s or (r|s)</td><td>Alternation string r or s</td></tr>
<tr><td>(r) |(s)</td><td>Alternation</td></tr>
<tr><td>[r]</td><td>An optional occurrence of string r</td></tr>
<tr><td>(r) *</td><td>>=0 occurrence of string r</td></tr>
<tr><td>(r)+</td><td>>=1 occurrence of string r</td></tr>
</table>

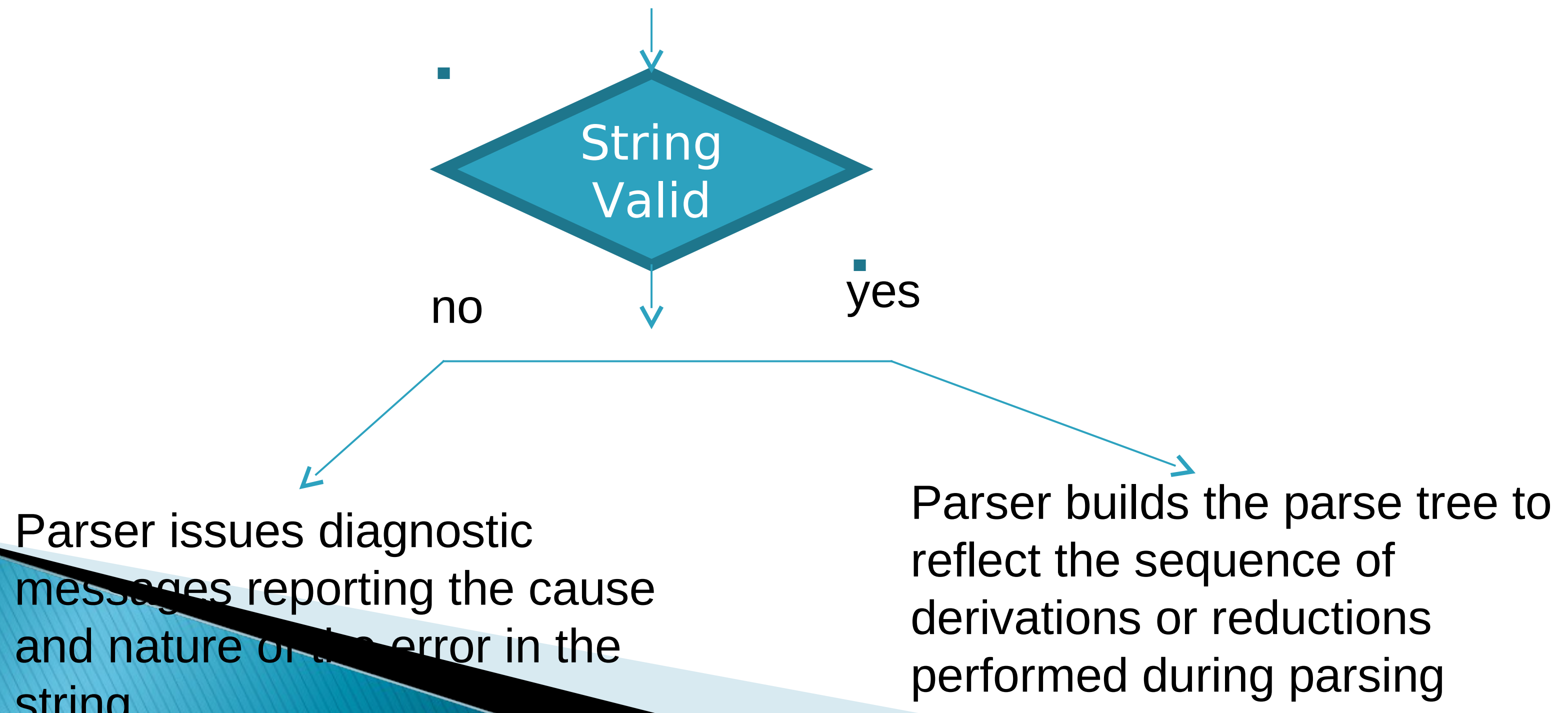
# Cont.



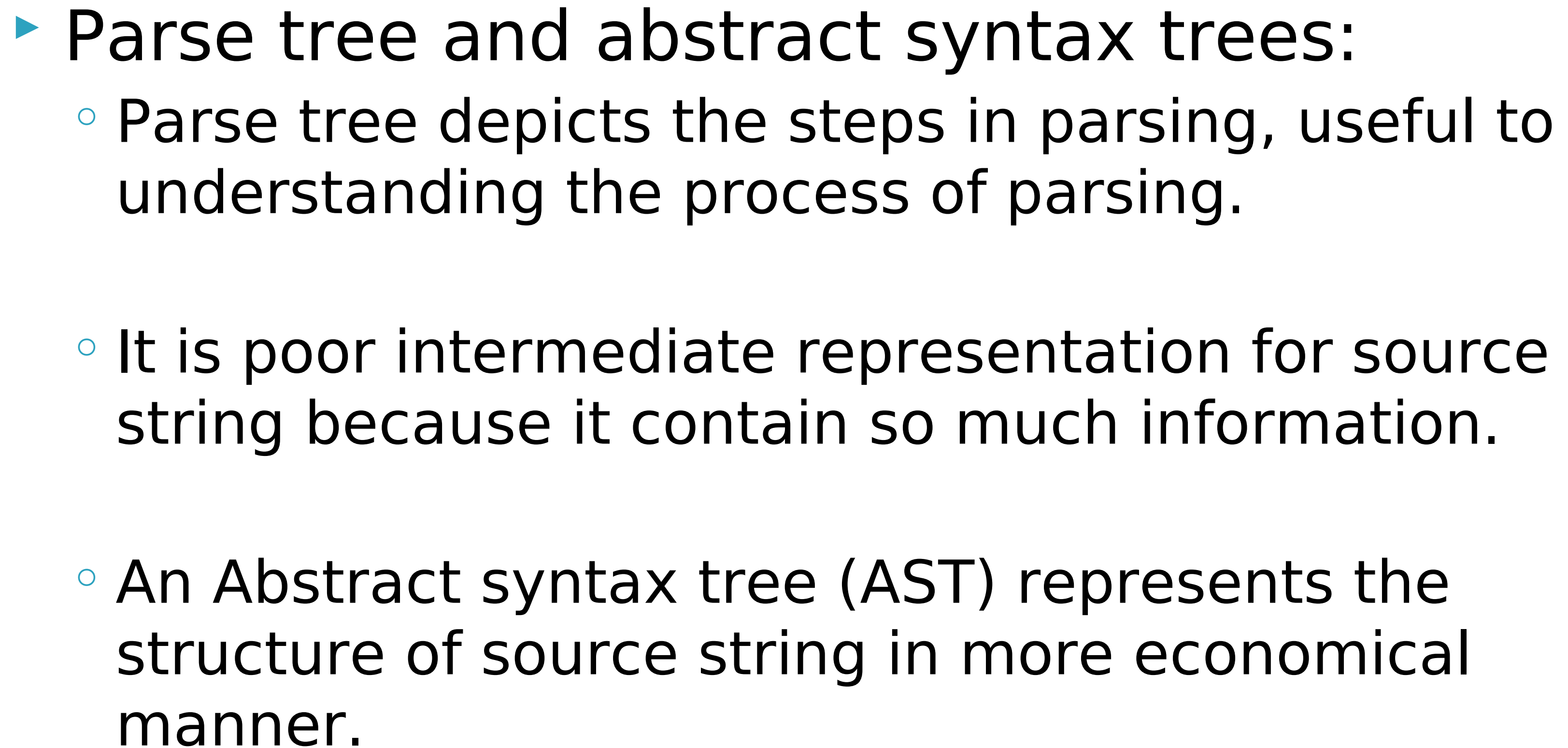
- Performing semantic actions:
  - Semantic action during scanning concern table building and construction of tokens for lexical components.
  - These action associated with the final states of DFA.

# Parsing

- - Why Parsing ?
  - To check the validity of a source string and to determine its syntactic structure.



# Cont.



- Parsing Methods
  - 1) Top down Parsing
  - 2) Bottom up Parsing

- Top Down Parsing
  - It tries to derive a string matching a source string through a sequence of derivation starting with the start symbol of the Grammar G.
  - Example for a valid string α
    - S => … => … => α

# Top- down parser

- Algorithm (Naïve top down parsing)
- Let α be the source String

1. Current sentential form (CSF)='S';
2. Let CSF be of the form βAΠ, such that β is string of Ts (note that β may be null), and A is the leftmost NT in CSF. Exit with success if CSF=α.
3. Make a derivation A=> β1Bγ according to a production A= β1Bγ of G such that β1 is a string of Ts (again β1 may be null). This makes CSF = ββ1BγΠ.
4. Go to step 2.

# Cont.

- **Continuation check** to determine whether the current sequence of derivations may be able to find a successful parse of α.
- CSF be of the form βAΠ, where β is a string of n Ts. All sentential forms derived from CSF would have the form β ____. For successful parse β must match the first n symbols of α.
- Also apply continuation check incrementally.

# Prediction and backtracking

- Prediction : this mechanism selects the RHS alternative of a production during prediction making. It must ensure that any String LG can be derived from S.
- Backtracking : This mechanism matches every terminal symbol generated during the derivation with the source symbol pointed to by the Source String Marker (SSM). If the match fails, Backtracking is performed,. This involves resetting CSF and SSM to earlier values.
- Example Lexically analysed version of the source string a+b*c.

<id> + <id> * <id> is to be parsed according to the following grammar

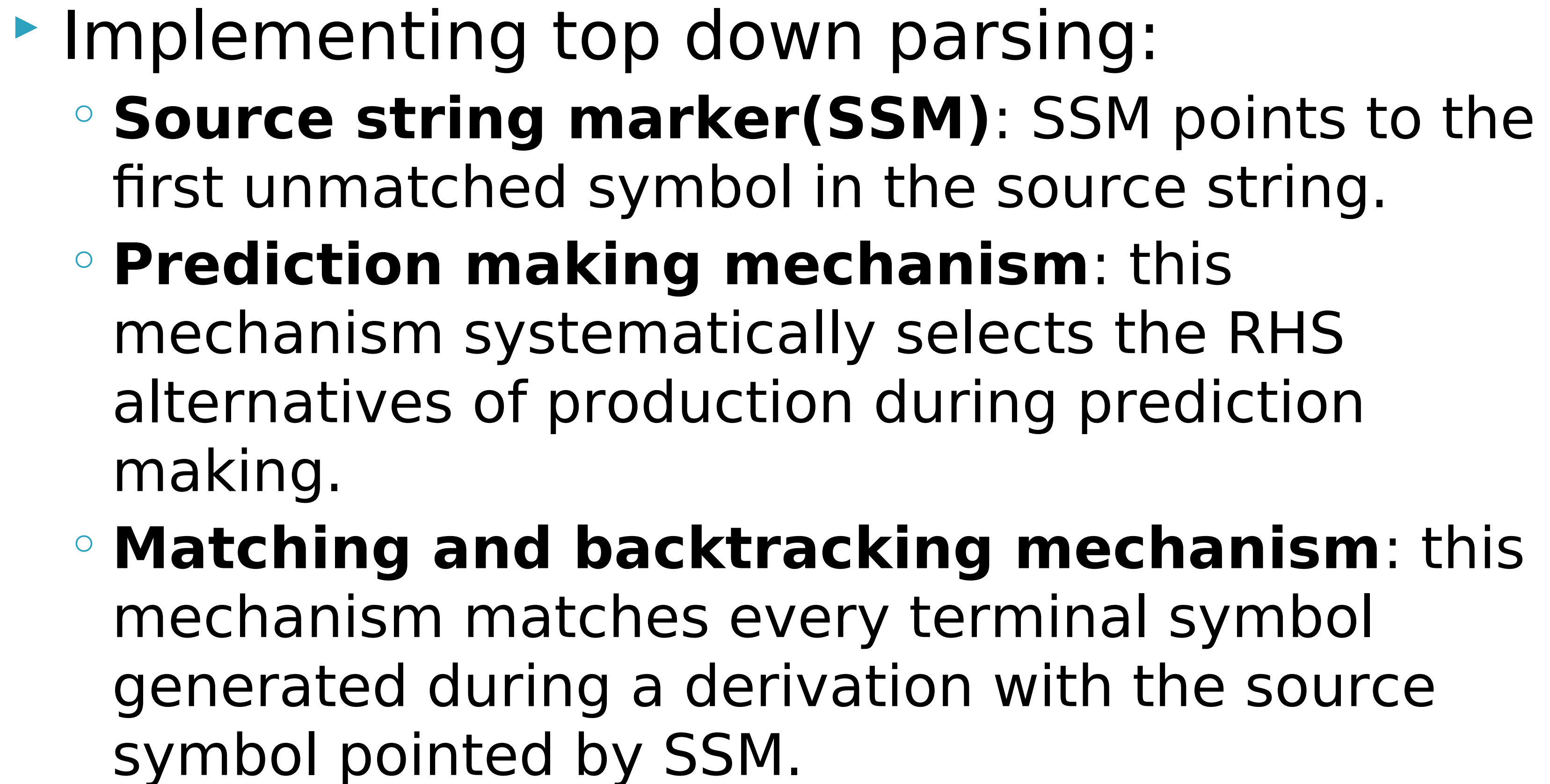
E ::= T + E | T

T ::= V * T | V

V ::= <id>

Here the prediction making mechanism selects the RHS alternative of a production in a left to right manner.

# Cont.



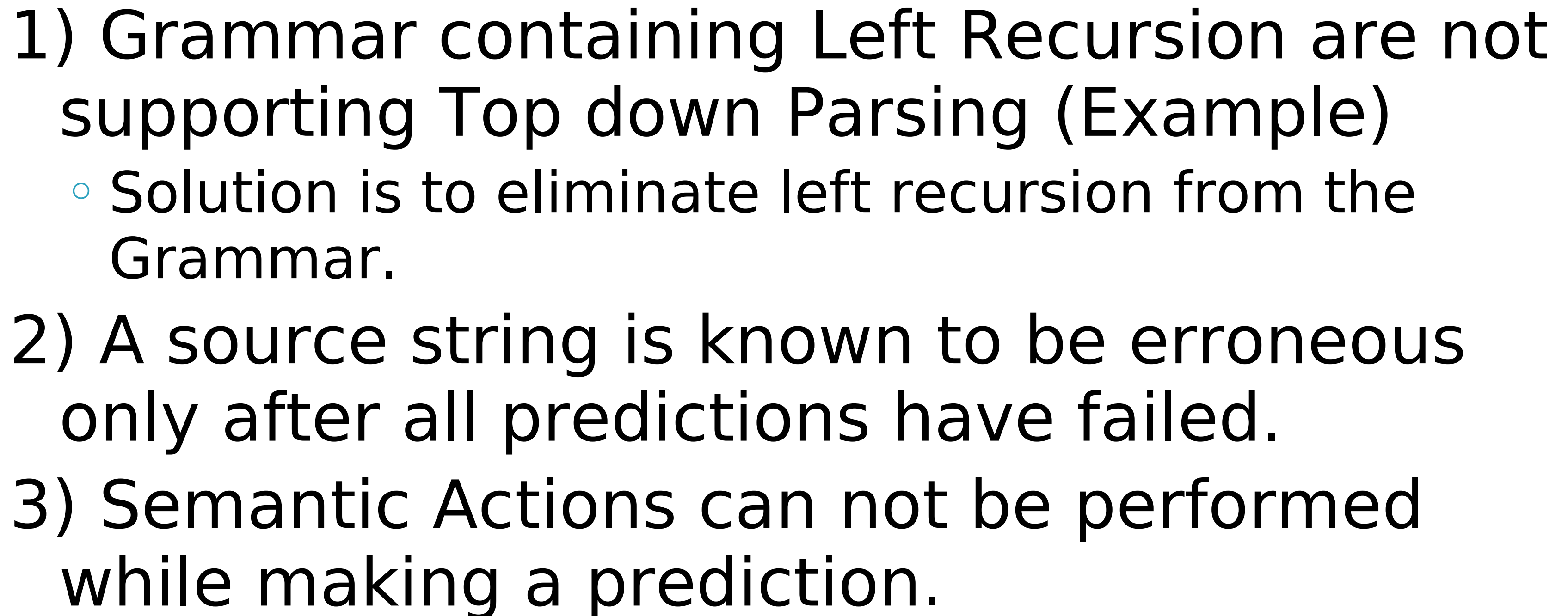
# Predictions and Backtracking

- SSM points to the first unmatched symbol in the source string
  <id> + <id> * <id>
- Let it be SSM := 1; CSF := E;
- Predictions in top down Parsing

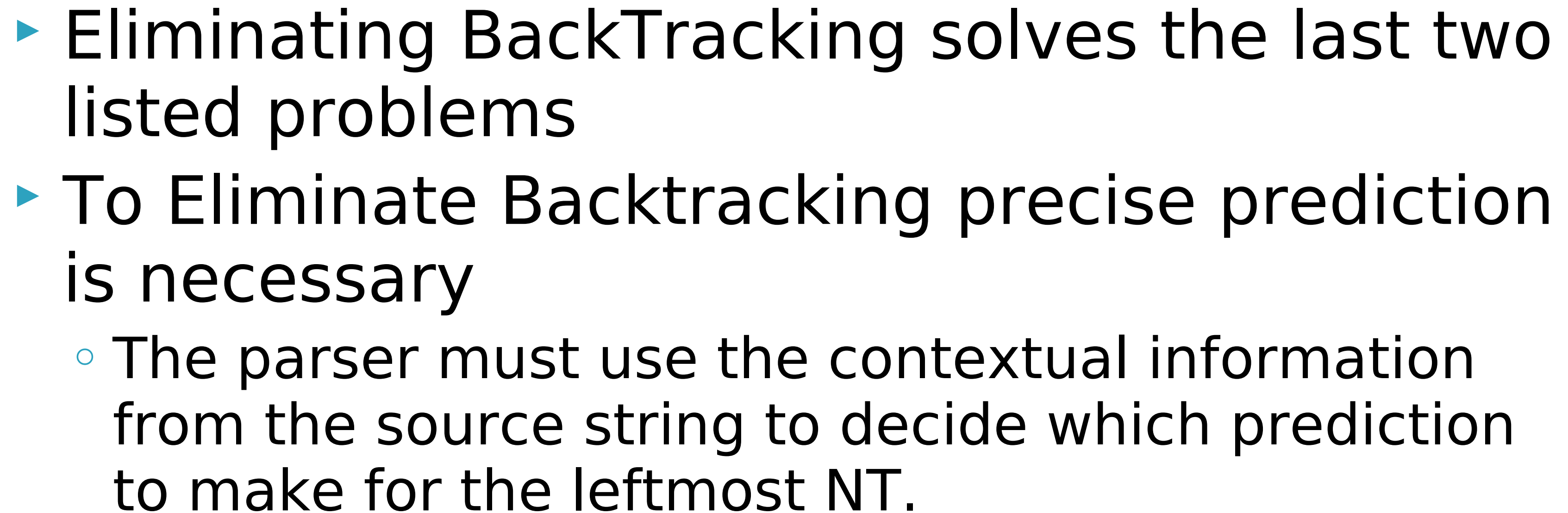
| | Prediction | Predicted Sentential Form | String to be matched | SSM | Matching Result |
|---|---|---|---|---|---|
| 1 | E=> T + E | T+E | | <id> | |
| 2 | T => V * T | V * T + E | | <id> | |
| 3 | V => <id> | <id> * T + E | | <id> | |
| | | | <id> | <id> matched | <id> matches with the first symbol of the source string. Thus SSM := SSM + 1 |
| | Match the second symbol of the prediction in Step 3 | | * | + | Fails thus reject Prediction shown at step 2 Thus the |

| Sr No | Prediction | Predicted Sentential Form | String to be matched from PSF | SSM | Matching Result |
|---|---|---|---|---|---|
| 1 | E=> T + E | T+E | | <id> | |
| 2 | T => V | V+E | | | |
| 3 | V => <id> | <id> + E | <id> | <id> | Matches thus SSM := SSM + 1 |
| | | | + | + | |
| 4 | | | | | Match Second Symbol of the PSF :Match Successful Thus SSM := SSM + 1 |
| | | | | <id> | |
| 5 | E => T | <id> + T | | | |
| 6 | T => V * T | <id> + V * T | | | |
| 7 | V => <id> | <id> + <id> * T | <id> | <id> | Match found Thus SSM := SSM + 1 |
| 8 | | | * | * | Match found Thus SSM := SSM + 1 |

# Problems with Top Down Parsing

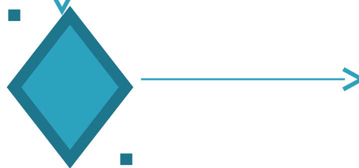


# Solution



If the left most NT is A and the source symbol pointed by SSM is t

Parser selects the RHS alternative of A which can produce ‘t’ as its first terminal symbol



Error if RHS alternative cant produce ‘t’ as the first terminal Symbol.

- Example
  - Consider Parsing String
  - <id> + <id> * <id>
  - The first prediction is to be made according to the Grammar
  - E::= T+E|T
  - Such that the first terminal symbol produced by it should be <id>
  - From the Grammar  we find that
  - T = > V …. And V => <id>
  - Thus any RHS alternative starting with T will produce <id>
  - However both alternative of E Start with T
  - Which alternative to Choose ?????

- Solution to this ambiguity is to use Left Factoring for the Grammar to Ensure that the RHS alternative will produce unique terminal symbol in the first position
- The production for E can be rewritten as
  - E ::= TE'
  - E' ::= +E|ϵ

- The first prediction according to grammar is
- E = > T E’ since the first source is <id>
- When E’ becomes the leftmost NT in CSF , the prediction E’ => +E is made if the next source symbol is + or else the prediction for E’ = ϵ is made
- Thus parsing is no longer by trial and error and no backtracking is performed This approach is called predictive Parsing.

- Example P2: Parsing of <id> + < id> * <id> according to following Grammar G3 ( Left Factoring is applied)
  - E ::= TE’
  - E’ ::= +E|ϵ
  - T ::= V T’
  - T’ ::= *T | ϵ
  - V ::= <id>

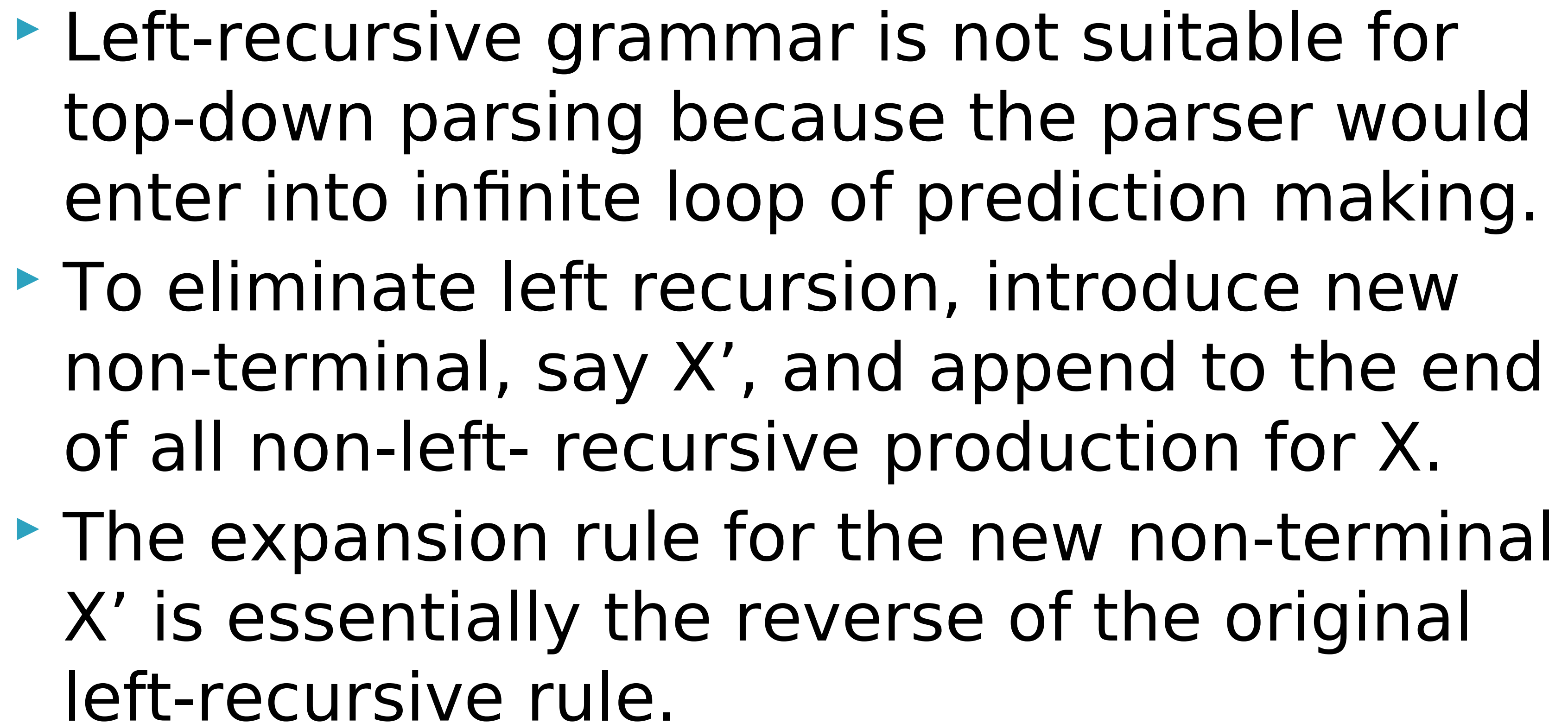
  Let CSF = E and SSM = <id> .

  Discuss this example of top down parsing without backtracking.

# Recursive Productions

- Grammar often has its productions defined in term of itself.
- “A non terminal on the LHS of the production occurs on the RHS as well, these are called recursive production.
- If recursive non-terminal occur on the left in the RHS of the production, A-> a|Ab,The production is said to be left recursive.
- The production has its recursive non-terminal on the right side on RHS,called A->a|bA is right recursive.

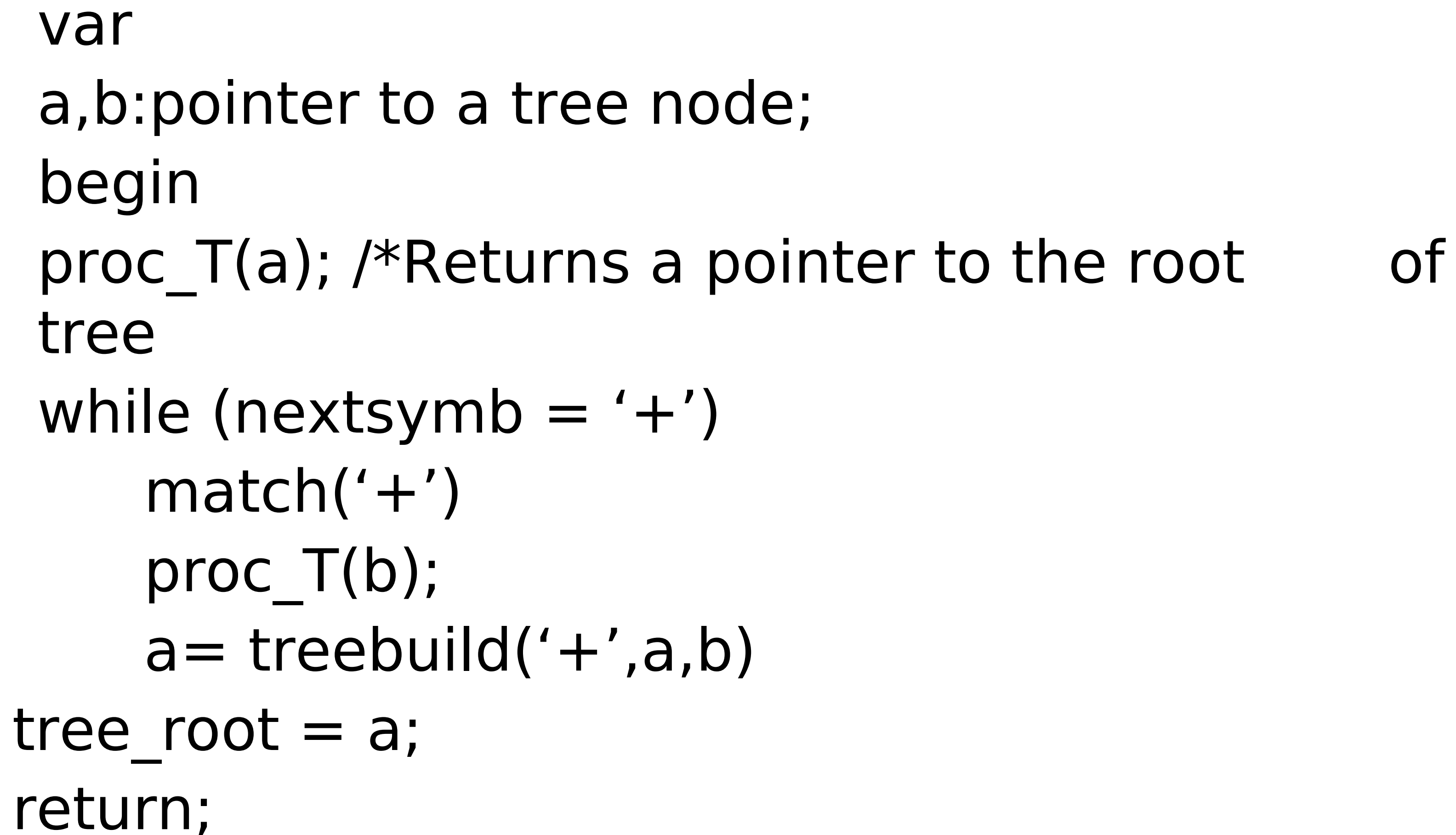
# Cont.



# A practical top down Parsing

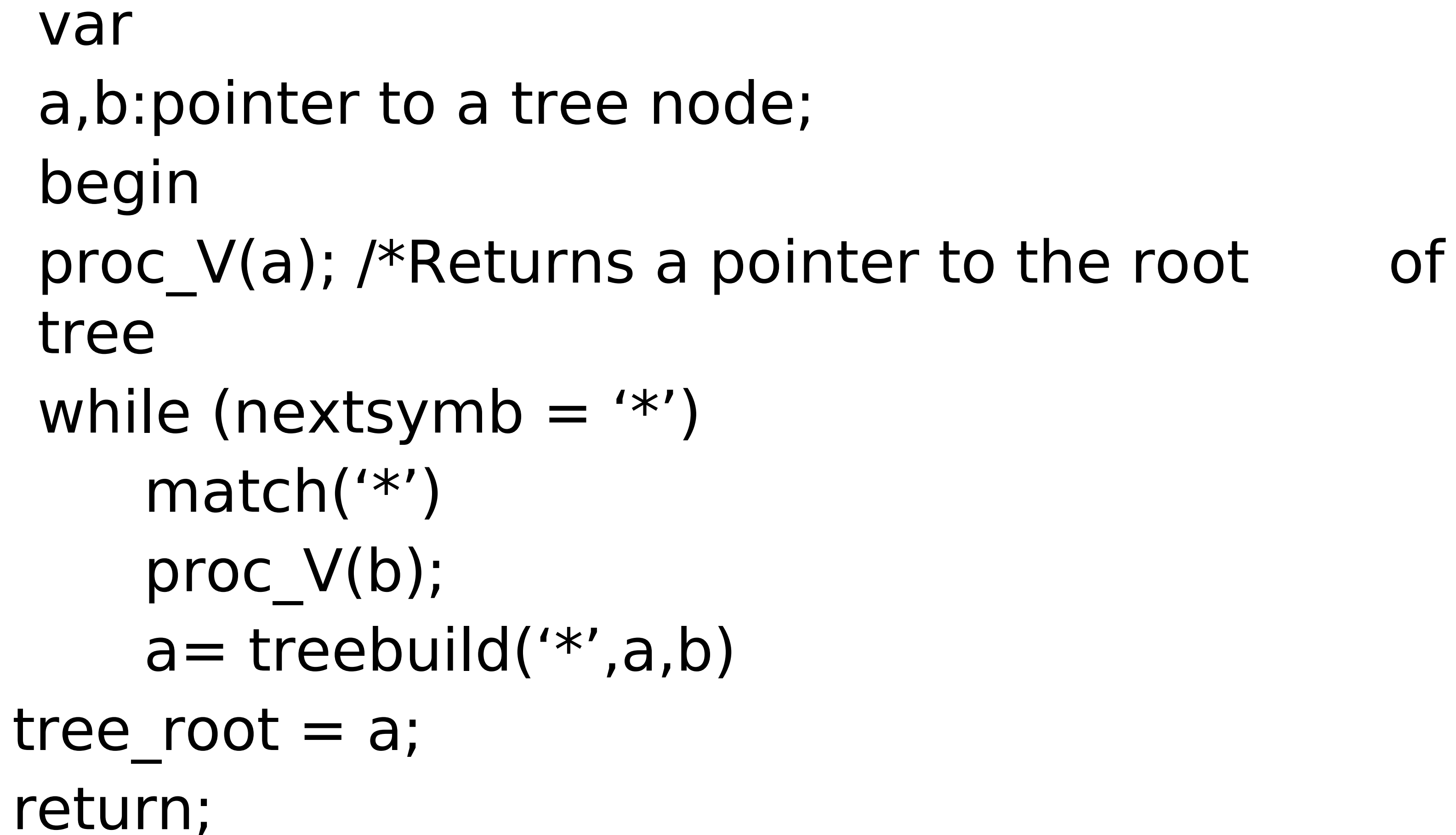
- Recursive descend (RD) parser is a variant of top down parsing without backtracking.
- It can be implemented any programming language which support Recursive procedure.
- To implement recursive descend parsing , a left factored grammar is modified to make repeated occurrences of string s more explicit.
- Grammar G3 is rewritten as
  - E ::= T { +T}*
  - T ::= V{* V}*
  - V ::= <id>

# Recursive Descent Parser Procedure proc_E( tree_root)



```
var
a,b:pointer to a tree node;
begin
proc_T(a); /*Returns a pointer to the root         of
tree
while (nextsymb = ‘+’)
      match(‘+’)
      proc_T(b);
      a= treebuild(‘+’,a,b)
tree_root = a;
return;
```

# Procedure proc_T( tree_root)



```
var
a,b:pointer to a tree node;
begin
proc_V(a); /*Returns a pointer to the root        of
tree
while (nextsymb = '*')
      match('*')
      proc_V(b);
      a= treebuild('*',a,b)
tree_root = a;
return;
```

# Procedure proc_V( tree_root)

```
var
     a:pointer to a tree node;
begin
if (nextsymb = <id>)
      match('id')
     tree_root= treebuild(<id>,-,-)
 else
     print "Error!"
return;
```

- LL(1) Parser
  - It is a table driven Parser for Left – to Left Parsing
  - ‘1’ indicates that the grammar uses a look ahead of one source symbol to make a prediction
  - The parsing table (PT)has a row for each NT and a column for each T ϵ Σ
  - A parsing table entry PT(nti , tj) indicates what prediction should be made if nti is the leftmost NT in a sentential form and tj is the next source symbol
  - A blank entry in this cell represents an error.
  - Parser starts with the sentential form |-E-|

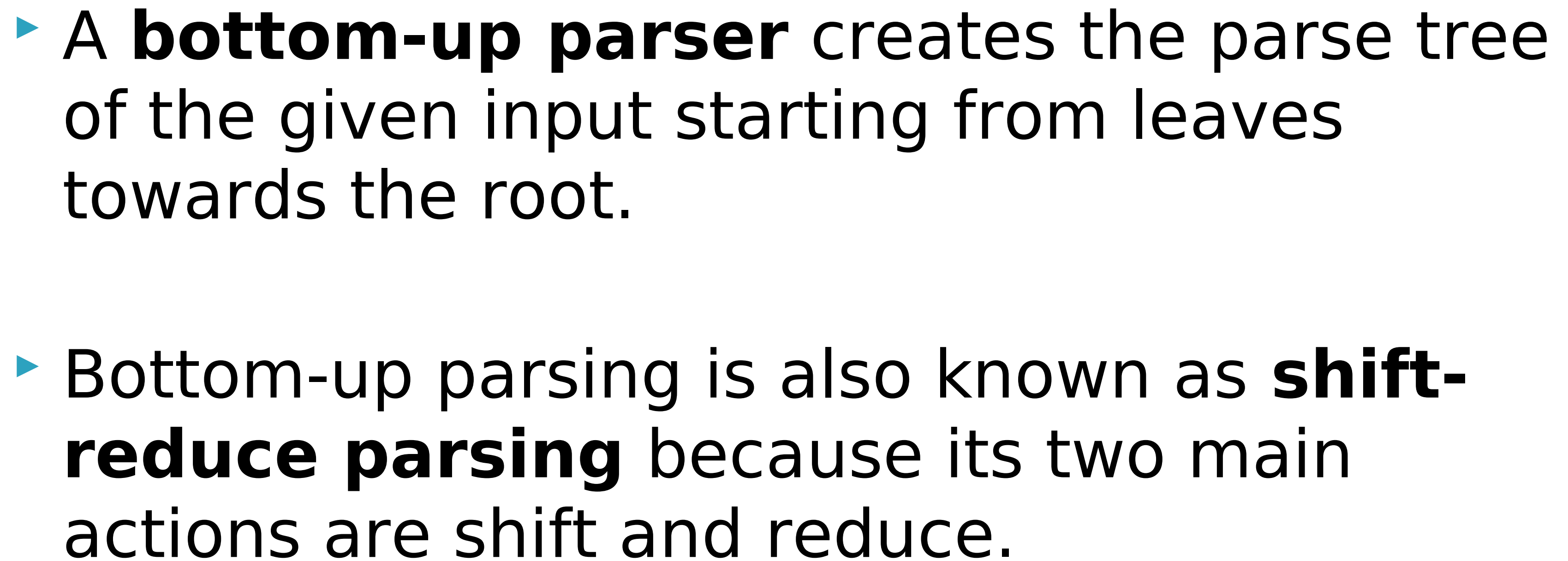
# Parsing Table

- The Parsing table for LL(1) parser for grammar G4
- E ::= TE’
- E’ ::= +TE’|ϵ
- T ::= VT’
- T’ ::= * VT’ | ϵ
- V ::= <id>

# Parsing Table

| Non Terminal (NT) | <id> | + | * | -\| |
|---|---|---|---|---|
| E | E => TE’ | | | |
| E’ | | E’ =>+TE’ | | E’=>ϵ |
| T | T=> VT’ | | | |
| T’ | | T’ => ϵ | T’=>*VT’ | T’ => ϵ |
| V | V=> <id> | | | |

# Bottom-Up Parsing



- A **bottom-up parser** creates the parse tree of the given input starting from leaves towards the root.

- Bottom-up parsing is also known as **shift-reduce parsing** because its two main actions are shift and reduce.

# Shift–Reduce Parsing

- A shift-reduce parser tries to reduce the given input string into the starting symbol.

  a string ➔ the starting symbol
  
  reduced to

- At each reduction step, a substring of the input matching to the right side of a production rule is replaced by the non-terminal at the left side of that production rule.
- If the substring is chosen correctly, the right most derivation of that string is created in the reverse order.

  Rightmost Derivation: $S \Rightarrow \omega$

  Shift-Reduce Parser finds: $\omega \Leftarrow ... \Leftarrow S$ rm

  rm

# Naïve Bottom up parsing

- 1. SSM =1;n=0;
- 2. r=n;
- 3. Compare the string of r symbols to the left of SSM with all RHS alternatives in G which have length of r symbols
- 4. If a match is found with a production A=α, then reduce the string of r symbols to the NT A.
-     n=n-r+1;
-     goto step 2;
- 5. r=r-1;
-      IF r >0 goto step3;

- 6. If no more symbols exit to the right of SSM then
-     if current string form = ‘S’ then
-         exit with success
-     else report error and exit with failure
- 7. SSM = SSM+1;
-    n=n+1;
-    goto step2;

# Shift-Reduce Parsing -- Example

S → aABb    input string:    aaabb

A → aA | a    aaAbb

B → bB | b    aAbb    ⇓ reduction

aABb

S

S ⇒ aABb ⇒ aAbb ⇒ aaAbb ⇒ aaabb

Right Sentential Forms

- How do we know which substring to be replaced at each reduction step?

# Handle

- Informally, a **handle** of a string is a substring that matches the right side of a production rule.
  - But not every substring matches the right side of a production rule is handle
- A **handle** of a right sentential form $\gamma$ ($\equiv \alpha\beta\omega$) is
  a production rule $A \rightarrow \beta$ and a position of $\gamma$
  where the string $\beta$ may be found and replaced by A to produce
  the previous right-sentential form in a rightmost derivation of $\gamma$.

  $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha A\omega \underset{rm}{\Rightarrow} \alpha\beta\omega$

- If the grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.
- We will see that $\omega$ is a string of terminals.

# Handle Pruning

- A right-most derivation in reverse can be obtained by **handle-pruning**.

$$S=\gamma_0 \overset{rm}{\Rightarrow} \gamma_1 \overset{rm}{\Rightarrow} \gamma_2 \overset{rm}{\Rightarrow} \ldots \overset{rm}{\Rightarrow} \gamma_{n-1} \overset{rm}{\Rightarrow} \gamma_n = \omega$$

input string

- Start from $\gamma_n$, find a handle $A_n \rightarrow \beta_n$ in $\gamma_n$, and replace $\beta_n$ in by $A_n$ to get $\gamma_{n-1}$.
- Then find a handle $A_{n-1} \rightarrow \beta_{n-1}$ in $\gamma_{n-1}$, and replace $\beta_{n-1}$ in by $A_{n-1}$ to get $\gamma_{n-2}$.
- Repeat this, until we reach S.

# A Shift-Reduce Parser

$E \rightarrow E+T \mid T$ Right-Most Derivation of id+id*id

$T \rightarrow T*F \mid F$ $E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*id \Rightarrow E+F*id$

$F \rightarrow (E) \mid id$ $\Rightarrow E+id*id \Rightarrow T+id*id \Rightarrow F+id*id \Rightarrow id+id*id$

| Right-Most Sentential Form | Reducing Production |
|---|---|
| <u>id</u>+id*id | $F \rightarrow id$ |
| <u>F</u>+id*id | $T \rightarrow F$ |
| <u>T</u>+id*id | $E \rightarrow T$ |
| E+<u>id</u>*id | $F \rightarrow id$ |
| E+<u>F</u>*id | $T \rightarrow F$ |
| E+T*<u>id</u> | $F \rightarrow id$ |
| E+<u>T*F</u> | $T \rightarrow T*F$ |
| <u>E+T</u> | $E \rightarrow E+T$ |
| E | |

<u>Handles</u> are red and underlined in the right-sentential forms.

# Operator Precedence Parsing

- Simple Precedence
  - A Grammar symbol a precedes b where each of a , b is a T or NT of Grammar G.
  - if in a sentential form …..ab…., ‘a’ should be reduced prior to ‘b’ in a bottom up parse then

  a precedes b can be written as

  $a \cdot> b$

- If a does not precede b then

  b precedes a written as $b \cdot> a$

- If a and b have equal precedence then it is represented as a = b.

# Simple Precedence Cont'd

- Where Precedence Relation can be defined ?
  - It can be defined between Grammar symbols a and b only if a and b can occur side by side in a sentential form

# How to obtain
# Simple Precedence Relation ?

- How to obtain Precedence Relation between a and b as follows
  1. Consider some sentential form …a,b…
  2. Determine the sequence of derivations

     S => …ab… such that the last derivation derives a string containing a or b or both. Number the derivations is this sequence from 1 to q.
  3. Consider the derivation numbered q. This is the last
     *
     derivation for obtaining …ab… Let this be of the form

     A => β  Then β -> A must be the first reduction in the bottom up parse of the string …ab…. Now

     a) a ·> b if β = …a

     b) a <· b if β = b…

     c a = b if β = …ab…

# Simple Precedence Grammar

- Grammar G is a simple precedence grammar if for all Terminal and nonterminal symbols a,b of G, a unique precedence relation exists for a,b

# Simple Precedence Grammar Cont'd

- In a SPG , a unit can be identified for reduction in a sentential form by looking to the precedence relations.

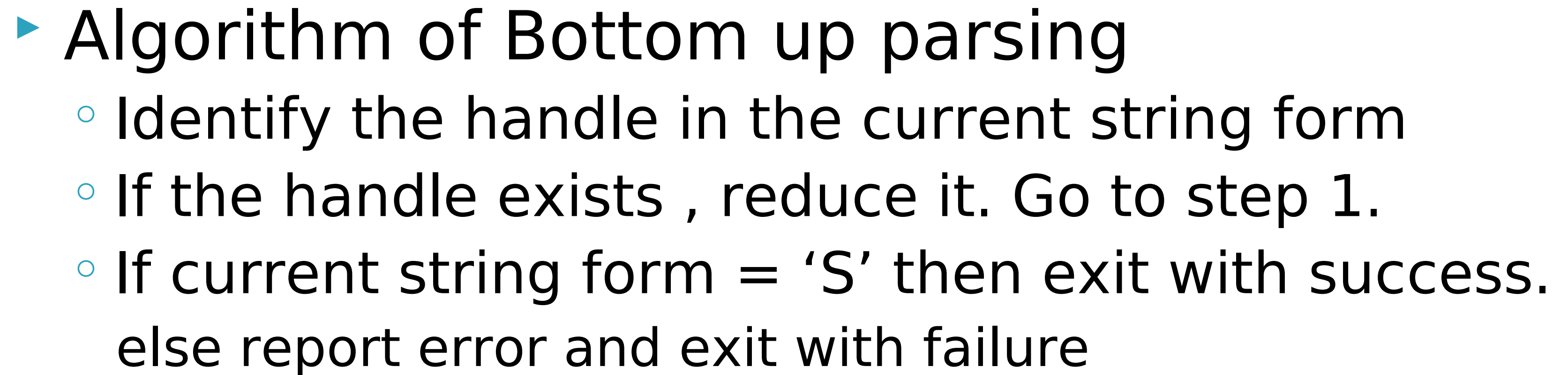
<· s1 = s2 = .... = sr ·> in the source string where each si is a Terminal or NT. Here s1,sr are the first and last symbols to participate in the reduction

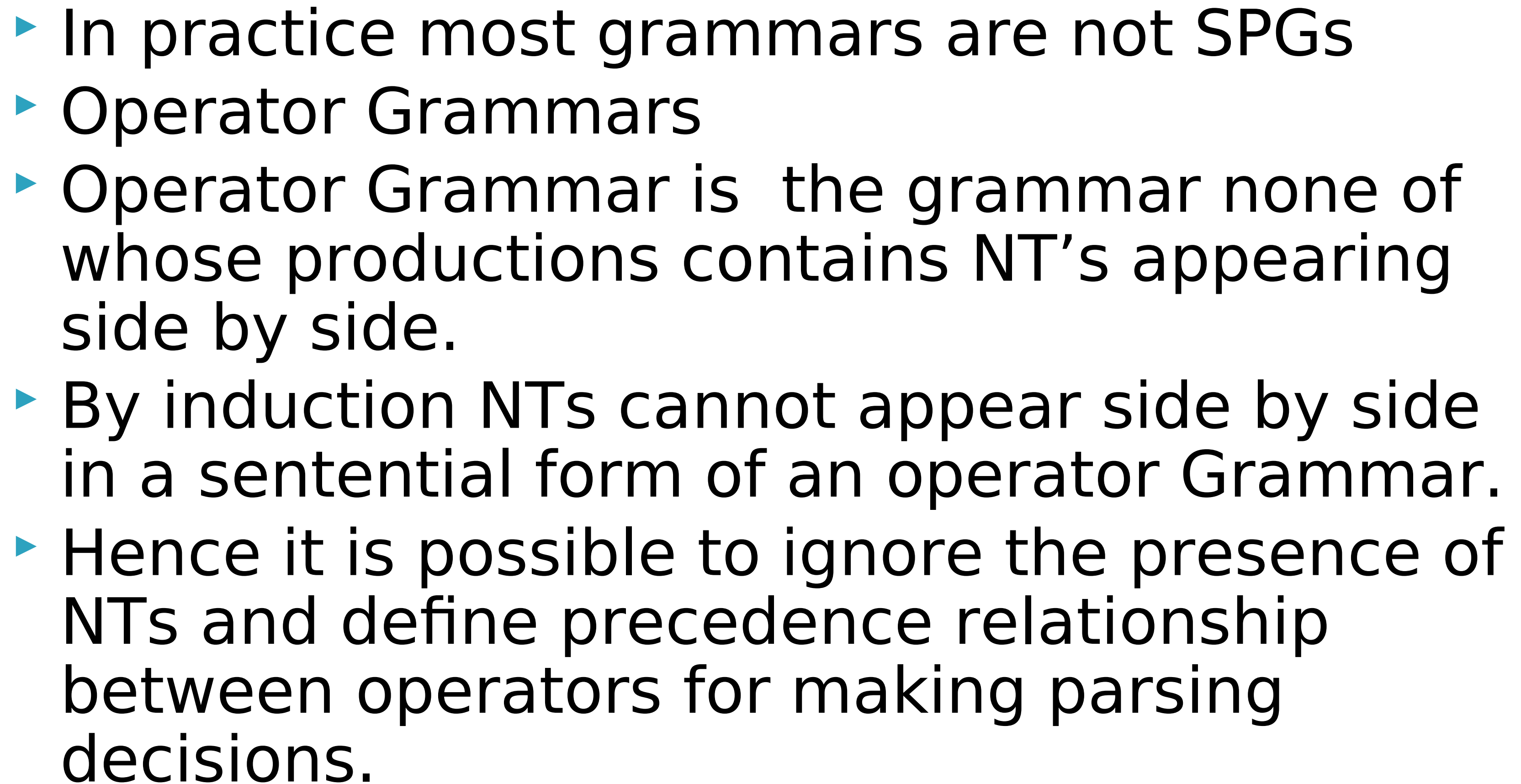
S1,s2…sr would form some RHS alternative in the Grammar.

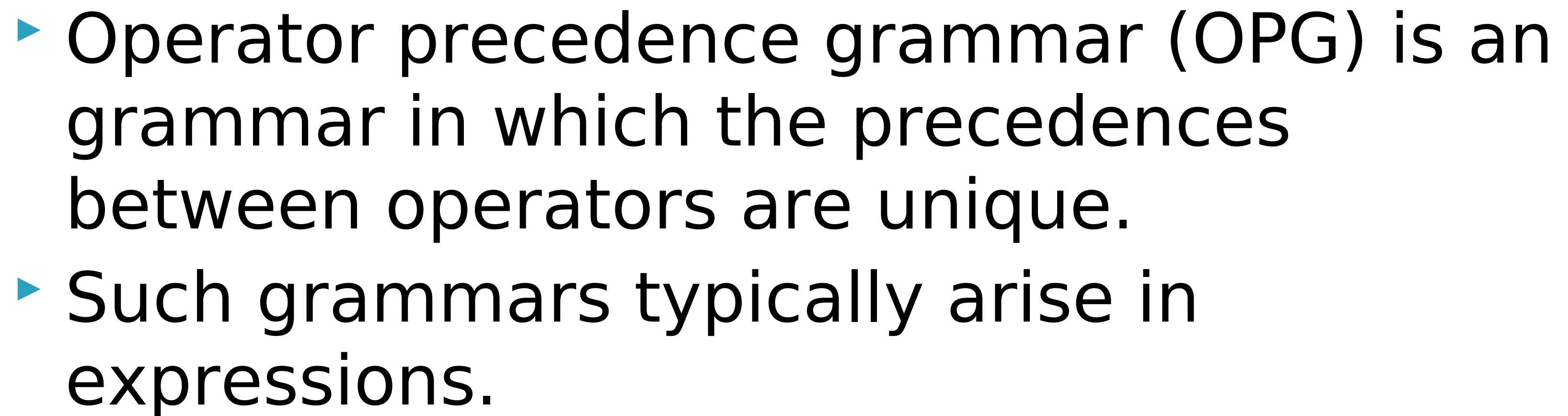
- Simple Phrase :

  α  is a simple phrase of the sentential form …αβ… if there exists a production of the grammar A ::= α and α -> A is a reduction in the sequence of reduction …αβ… -> …-> S.
- Handle : A handle of a sentential form is the left most simple phrase in it.

- E + T is not a simple phrase of the sentential form E+T*F according to grammar 1 since it’s reduction to E does not lead to distinguished symbol. However T * F is a simple phrase of the sentential form
- Deficiency of the Bottom up parsing algorithm is in its failure to identify simple phrases and handles.


Algorithm of Bottom up parsing
Identify the handle in the current string form
If the handle exists , reduce it. Go to step 1.
If current string form = ‘S’ then exit with success.
else report error and exit with failure


In practice most grammars are not SPGs
Operator Grammars
Operator Grammar is the grammar none of whose productions contains NT’s appearing side by side.
By induction NTs cannot appear side by side in a sentential form of an operator Grammar.
Hence it is possible to ignore the presence of NTs and define precedence relationship between operators for making parsing decisions.



- Operator precedence grammar (OPG) is an grammar in which the precedences between operators are unique.
- Such grammars typically arise in expressions.

# What is operator precedence ?

- Precedence between operators a and b appearing in a sentential form ...aPb... where P is an NT or  a null string is termed operator precedence.

# Rules to determine operator precedence.

- Rules to determine precedence relations from the productions of G is as follows
- 1) a = b if there exists a grammar production

  C ::= βabγ   or C::= βaAbγ

  2) a ·> b if there exists grammar productions

  C ::= βAbγ  and A ::= πa | πaD

  3) a <· b if there exists grammar productions

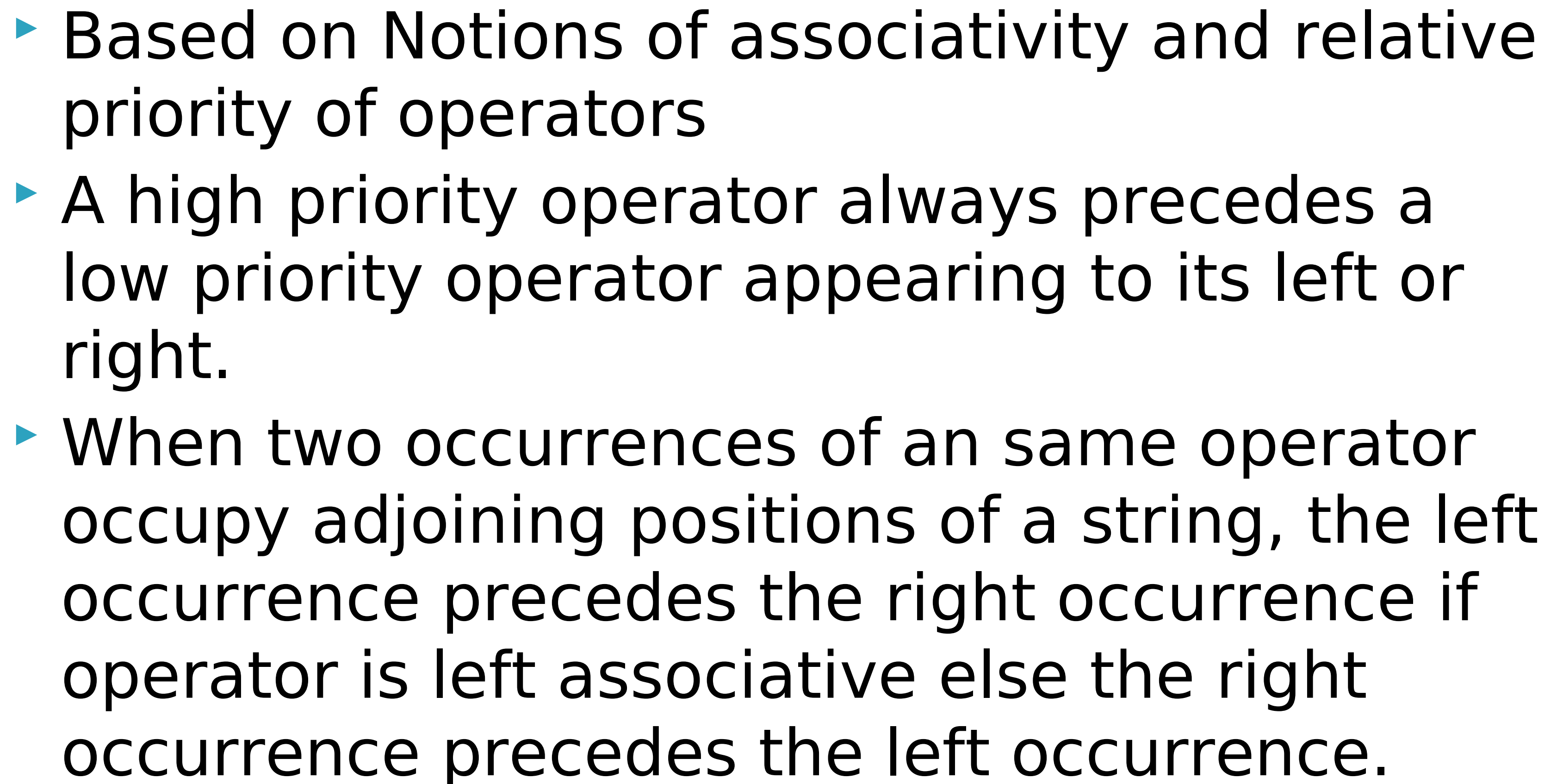
  C ::= βaBγ and

  B ::= bδ | Dbδ

# Operator Precedence Matrix

- OPM represents operator precedence relations between pair of operators.
- The entry OPM(a,b) represents the precedence of operator a with respect to operator b in a sentential form …aPb…, where P may be a null String.

# Simpler way to obtain precedence relations



- Consider the following Grammar to parse the following string

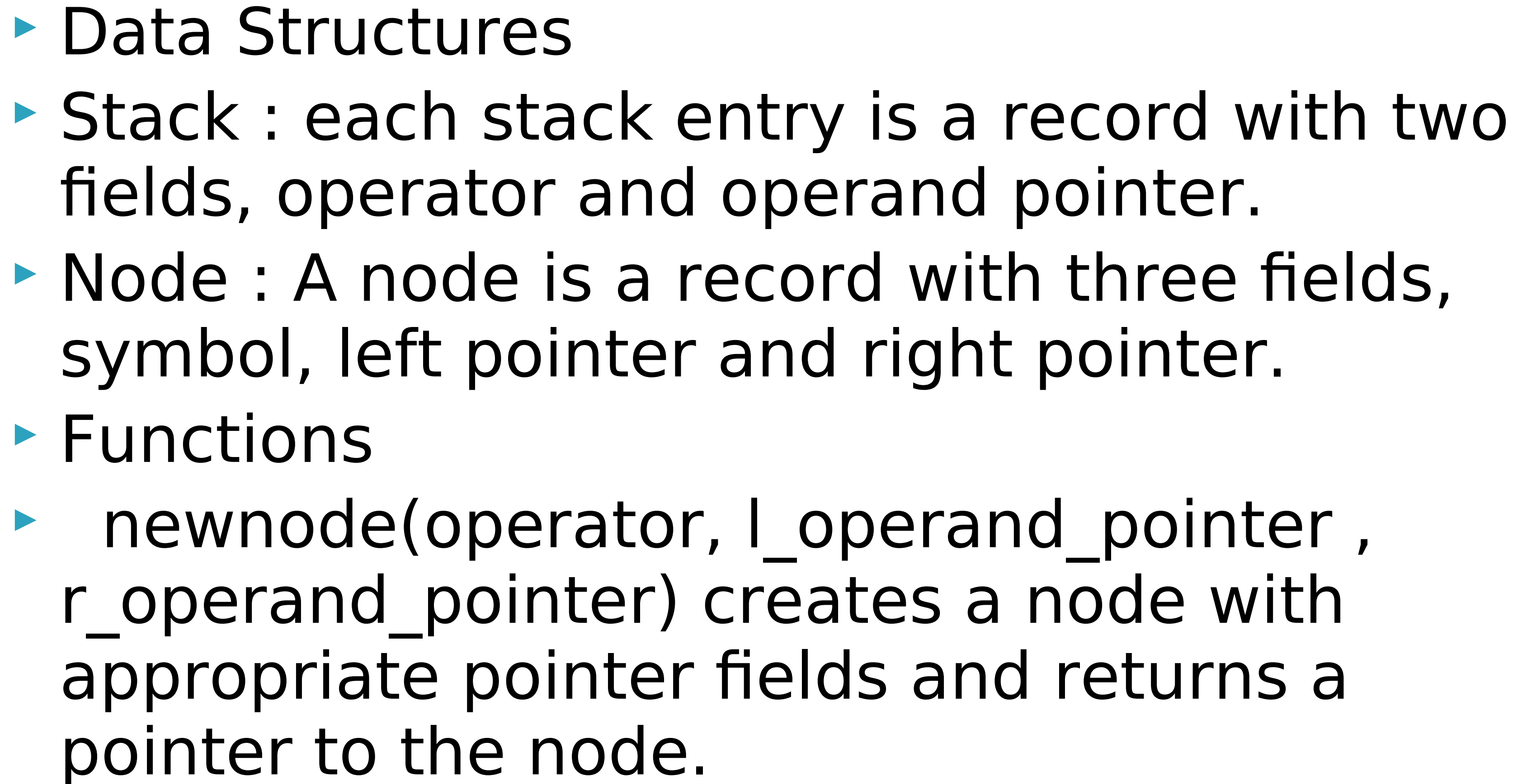
  |- <id> + <id> * <id> -|

  Grammar
- S ::= |- E -|
- E ::= E + T | T
- T ::= T * V | V
- V ::= <id> | (E).

# Operator Precedence Matrix

| **LHS Operator** | **RHS operators** | | | | |
|---|---|---|---|---|---|
| | + | * | ( | ) | -\| |
| + | ·> | <· | <· | ·> | ·> |
| * | ·> | ·> | <· | ·> | ·> |
| ( | <· | <· | <· | = | |
| ) | ·> | ·> | | ·> | ·> |
| \|- | <· | <· | <· | | = |

# Operator Precedence Parsing Algorithm.



- Data Structures
- Stack : each stack entry is a record with two fields, operator and operand pointer.
- Node : A node is a record with three fields, symbol, left pointer and right pointer.
- Functions
- newnode(operator, l_operand_pointer , r_operand_pointer) creates a node with appropriate pointer fields and returns a pointer to the node.

1 ) TOS := SB – 1 ; SSM:= 0

2) Push |- on the stack

3) SSM = SSM +1;

If current source symbol is an operator then go to step 5

4) X = newnode(source_symbol,null,null);

Tos.operand_pointer := x;

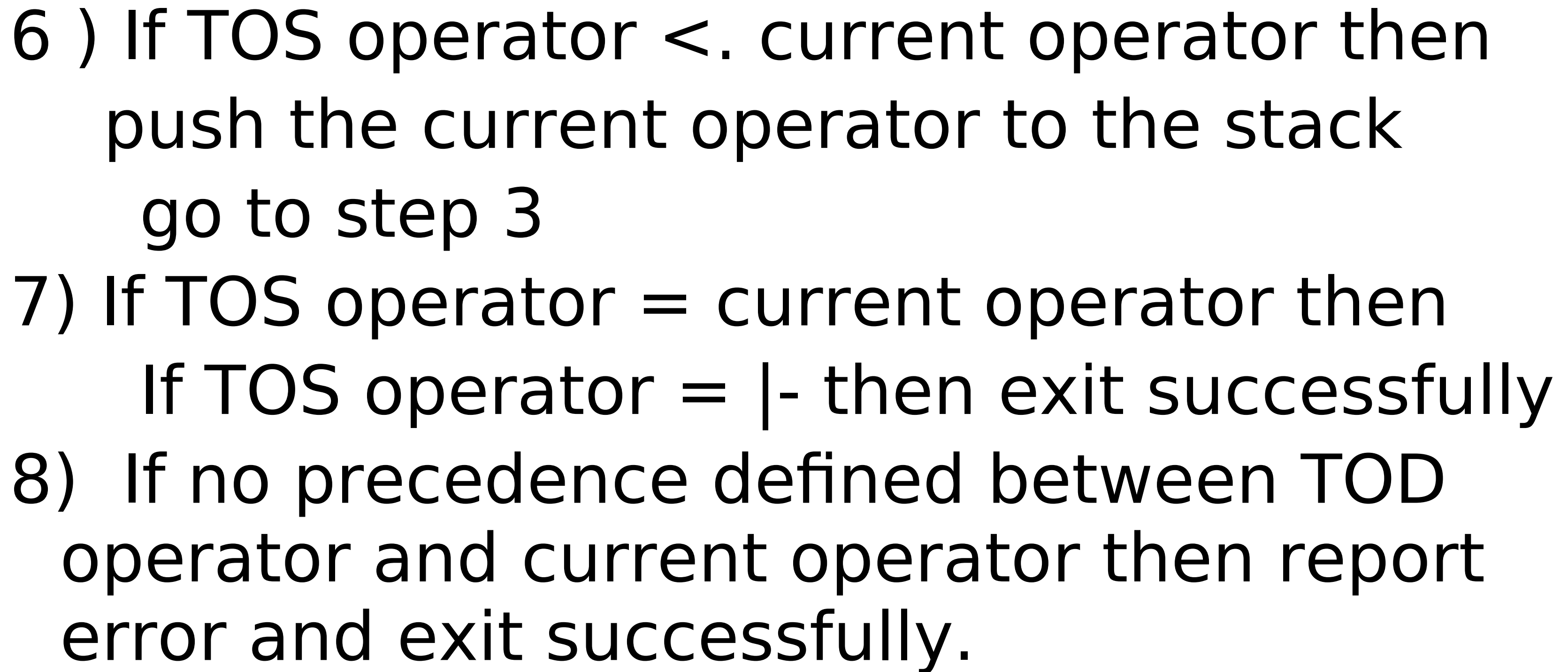
Go to step 3;

5) While TOS operator .> current operator

x := newnode(TOS operator, TOSM.operand_pointer, Tos.operand_pointer);

pop an entry off the stack

TOS.operand_pointer = x;


6 ) If TOS operator <. current operator then
push the current operator to the stack
go to step 3
7) If TOS operator = current operator then
If TOS operator = |- then exit successfully
8) If no precedence defined between TOD operator and current operator then report error and exit successfully.