Unit 6
Top down backtrack parsing

Problem of parsing

Given a Context Free Grammar G and a string w.

is w generated by G?

Most parsing methods fall into one of two classes: top down and bottom up, depend on the order in which nodes in the parse tree are constructed

W has been parsed ⇒parse tree is constructed

E -> E + T

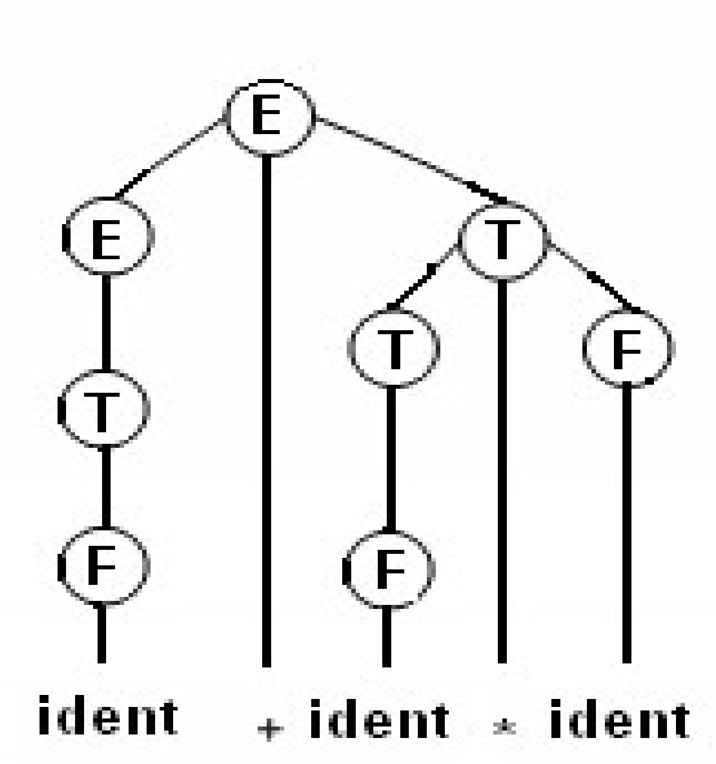
E -> T

T -> T * F

T -> F

F -> (E)

F -> ident



How to express parse trees ?

Left parse

- *Left parse of* α is the sequence of productions used in left derivation of α from S

- *Left parse is a string with numbers from 1 to p*

Example

- Consider grammar G, with productions are numbered

  1. $E \rightarrow T+E$
  2. $E \rightarrow T$
  3. $T \rightarrow F*\ T$
  4. $T \rightarrow F$
  5. $F \rightarrow (E)$
  6. $F \rightarrow a$

- Left parse of string a*(a+a) is 23645146246

Top down backtrack parsing algorithm

- Input: Grammar G, string w
- Output one left parse for w if one exists. The output "error" otherwise

- For each nonterminal A , order the alternate for A : $A \rightarrow \alpha_1 \mid \alpha_2 \mid . . . . \mid \alpha_n$

Description of algorithm

- Begin with a tree containing one node labeled S
- S is considered active node
- Other nodes are generated recursively

Active node is label by nonterminal A

- Choose the first alternate of A: $X_1X_2. . . .X_k$.
- Create k direct descendants for A labeled X_1, X_2, ...X_k.
- Make X_1 the active node
- If k = 0, (production A → ε) make the node immediately to the right pf A active

Active node is labeled by a terminal *a*

- Compare the current input symbol with a.
  - □ If they match, make the active node immediately to the right of *a*, move the input pointer one symbol to the right.
  - □ If they do not match, go back to the node where the previous production was applied.
  - □ Adjust the input pointer if necessary, and try the next alternate. If no alternate is possible, go back to the next previous node, and so forth.
- If the current node is the root and no alternate is possible, emit an error message.

A stringent condition

- *Grammar G must be non left recursive to avoid a non-termination*

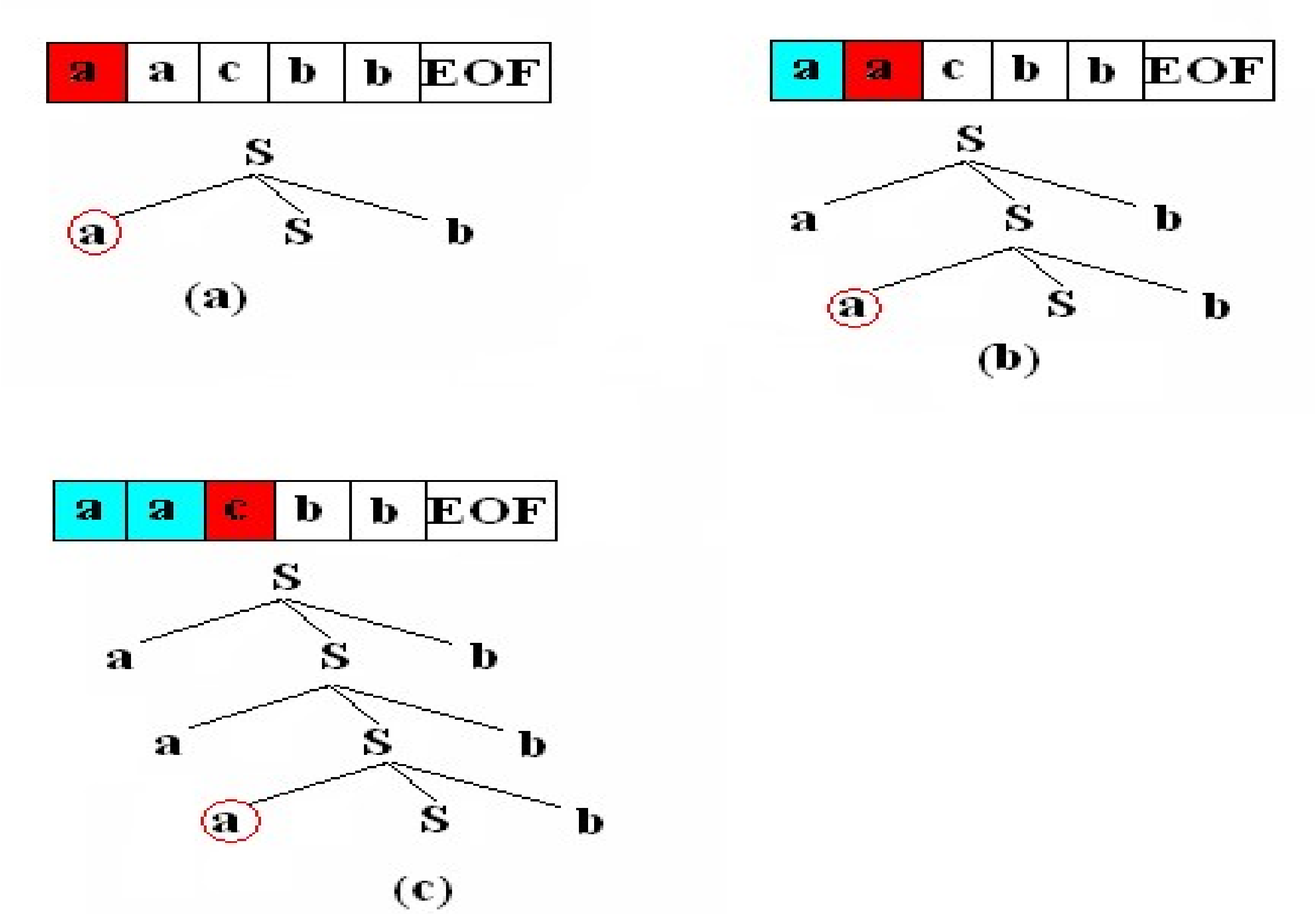
Example

- Given grammar

  $$S \rightarrow aSb \mid c$$

  productions are numbered from 1 to 2.
- And string w= aacbb

Build parse tree

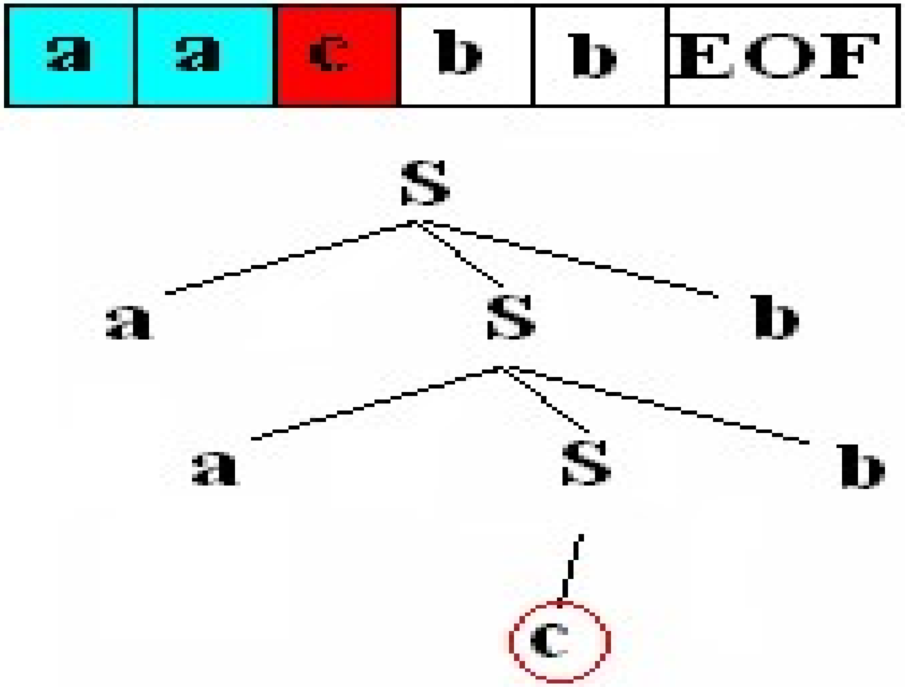


(a)

(b)

(c)

Try another alternate



(d)

The top down parsing algorithm

- **Input**

  A non-left recursive context free grammar G,
  Input string $w = a_1. . . .a_n$, $n \geq 0$
  Assume the productions in P are numbered 1,. . . q
- **Output**

  One left parse for w if one exist.
  “Error” otherwise.

Method

- *(The algorithm uses 2 stacks D_1 and D_2).*

D_2 represents the current left sentential form, which our expansion of nonterminals has produced.

D_1 represents the current history of the choices of alternates made and the input symbols over which the input head has shifted.

(1)

- $\forall$ A $\in$ N , if all af A-productions in P are

  $A \to \alpha_1 \mid \alpha_2 \mid . . . . \mid \alpha_n$

Order the alternates

$A_1 \to \alpha_1$

. . . .

$A_n \to \alpha_n$

Configuration of the algorithm

4-tuple (s, i, α, β)

- s $\in$ Q: the current state
  - q: normal operation
  - b: backtracking
  - t: terminating
- i : location of the input pointer (the n+1 st "input symbol" is #, the right endmarker)

  α: content of the first stack (D1)

  β: content of the second stack (D2)

Execution of the algorithm

- Starting in the initial configuration, compute successive next configurations until no further configuration can be computed.

- If the last computed configuration is (t,n+1,γ,ε), emit h(γ) and halt. Otherwise emit the error signal.

Example

- Consider string aacbb and grammar G

$$S \rightarrow aSb$$
$$S \rightarrow c$$

Number productions

1. $S_1 \rightarrow aSb$
2. $S_2 \rightarrow c$

Sequence of configurations

$(q, 1, \varepsilon, S\#)$

$\vdash (q, 1, S_1, aSb\#)$

$\vdash (q, 2, S_1a, Sb\#)$

$\vdash (q, 2, S_1aS_1, aSbb\#)$

$\vdash (q, 3, S_1aS_1a, Sbb\#)$

$\vdash (q, 3, S_1aS_1aS_1, aSbbb\#)$

$\vdash (b, 3, S_1aS_1aS_1, aSbbb\#)$

$\vdash (q, 3, S_1aS_1aS_2, cbb\#)$

$\vdash (q, 4, S_1aS_1aS_2c, bb\#)$

$\vdash (q, 5, S_1aS_1aS_2cb, b\#)$

$\vdash (q, 6, S_1aS_1aS_2cbb, \#)$

$\vdash (t, 6, S_1aS_1aS_2cbb, \varepsilon)$

Recover the left parse

- *$h(a) = \varepsilon$ for all terminal a*
  $h(A_i)= p$,
  p is the production number associated with the production $A \to \gamma$ and γ is ith alternate for A
- *Example : with grammar*
  1. $S_1 \to aSb$
  2. $S_2 \to c$
- $h(S_1aS_1aS_2cbb)=112$

Top-down parser with bactraking for KPL

- Scan the stream and find tokens
- Set of production

From syntax diagrams to BNF

<program >::= *program ident ;* <block>.

<block>::= <const-decl><type-decl>

<proc-decl><func-decl><var-decl> *begin* <statement-list> *end*

Non-terminal encoding

```
if(str=="<program>") return 1;
if(str=="<block>") return 2;
if(str=="<const-decl>") return 3;
if (str == "<const-assign-list>") return 4;
if (str == "<constant>") return 5;
if(str=="<type-decl>") return 6;
if (str =="<type-assign-list>") return 7;
if (str == "<type>") return 8;
if (str == "<basictype>") return 9;
if(str=="<var-decl>") return 10;
if (str == "<ident-list>") return 11;
if(str=="<proc-decl>") return 12;
if (str == "<para-list>") return 13;
if (str == "<para-one>") return 14;
if(str=="<func-decl>") return 15;
if(str=="<statement-list>") return 16;
if(str=="<statement>" ) return 17;
if (str == "<condition>") return 18;
if (str == "<relation>") return 19;
if(str=="<expression>") return 20;
if (str == "<adding-op>") return 21;
if(str=="<term>") return 22
if(str=="<multiplying-op>") return 23;
if (str == "<factor>") return 24;
```

Token encoding: identifiers, number, character constants, specific symbols

```
// ident;
 if(str == "ident") return 25;
//const
 if(str == "number")return 26;
 if (str == "charcon") return 27;
//operator
 if(str == "plus")return 28;
 if (str == "minus") return 29;
 if (str == "times") return 30;
 if (str == "slash") return 31;
 if (str == "assign") return 33;
 if (str == "leq") return 34;
```

```
//specific symbol
if (str =="lparen") return 35;
if (str == "rparen") return 36;
if (str == "comma") return 37;
if (str == "semicolon") return 38;
if (str == "period") return 39;
if (str == "becomes") return 40;
if (str == "lbrace") return 41;
if (str == "rbrace") return 42;
if (str == "lbrack") return 43;
if (str == "rbrack") return 44;
```

Token encoding: keywords

```
if (str == "beginsym") return 45;
if (str == "endsym") return 46;
if (str == "ifsym") return 47;
if (str == "thensym") return 48;
if (str == "whilesym") return 49;
if (str == "dosym") return 50;
if (str == "callsym") return 51;
if (str == "constsym") return 52;
```

```
if (str == "varsym") return 53;
if (str == "progsym") return 54;
if (str == "funcsym") return 55;
if (str == "typesym") return 56;
if (str == "arraysym") return 57;
if (str == "ofsym") return 58;
if (str == "intsym") return 59;
if (str == "charsym") return 60;
```

Token encoding : relop

```
//relations
if (str == "eql") return 61;
if (str == "leq") return 62;
if (str == "neq") return 63;
if (str == "lss") return 64;
if (str == "gtr") return 65;
if (str == "geq") return 66;
```

Production encoding

<program >::= *program ident ;* <block>.

setlaw[1,1]="54 25 38 2 39 ";

<block>::= <const-decl><type-decl>
<proc-decl><func-decl><var-decl> *begin* <statement-list> *end*

setlaw[2,1]=" 3 6 12 15 10 45 16 46 ";

Conclude

- Too complicated with backtracking.
- Spent exponential amount of time.
- Difficult to handle errors.