Recursive and Recursive descent parser

Dr G Sudha Sadasivam

Recursive Procedures

- Every NT “A” has a procedure. RHS of the NT “A” specifies the structure of the code for the procedure.
- The sequence of terminals on the right hand side corresponds to input matches,
- Sequences of non-terminals are calls to the corresponding procedures.
- S-->cAd   A-->ab | a

isave saves the input pointer position

in-ptr advances to the next position using **Advance**()

Parser returns a "**true**" value on successful completion

S-->cAd A-->ab |a

```
procedure S( )
{
    if input = 'c'
    {
    Advance( )
    A( );
    if input = 'd'
    {
    Advance( );
    return true;
    }
    else
    return false;
   else
      return false;
   }
```

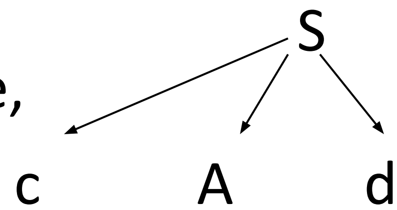
```
procedure A( )
{
    isave=in-ptr;
    if input = 'a'
    {
        Advance();
        if  input = 'b'
        {
              Advance( );
              return true;
        }
    }
    in-ptr=isave
    if input='a'
    {
            Advance( );
            return true;
    }
    return false;
}
```

Issues in Recursive parsing

1. **Left Recursion:** In left recursion the grammar has productions of the format **$A \longrightarrow A\alpha$**
2. **Backtracking:** It occurs when there is more than one alternate in the productions to be tried while parsing the input string $A \longrightarrow ab \mid a$
3. It is very difficult to identify **the position of the errors.**

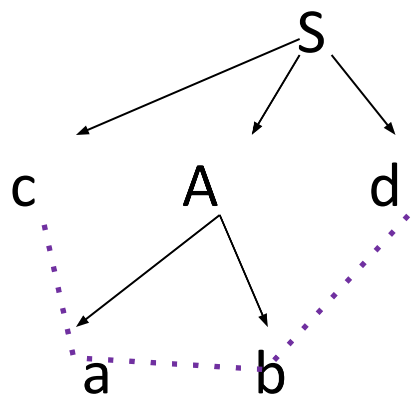
Recursive parsing with backtracking : example to derive string cad

Following the first rule, S->cAd to parse S

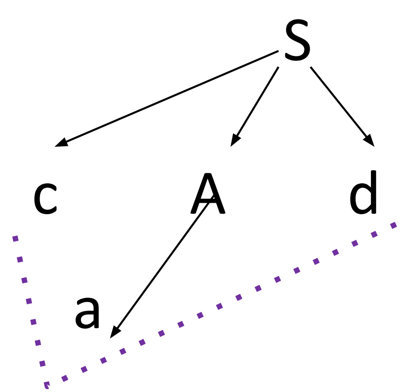


S->cAd

The next NT "A" is parsed using first rule, A -> ab , but turns out INCORRECT, parser backtracks



A -> ab



A -> a

Next rule to parse A is taken A->a, turns out CORRECT , Parser stops

Left factoring – to eliminate backtracking

- Left factoring is a grammar transformation to eliminate backtracking and ambiguity
- When it is not clear which of two alternative productions to use to expand a non-terminal A (like A□ ab | a), defer the decision until we have seen enough of the input to make the right choice like

  $A \longrightarrow a\ A'$ and $A' \text{-->}\ b \mid \epsilon$

Example: $A \text{ -> } xP_1 \mid xP_2 \mid xP_3 \mid xP_4 \ldots\ldots \mid xP_n$

Where x & Pi's are strings of terminals and non-terminals and $x \mathrel{!=} \varepsilon$

In left removing, we rewrite it as – to remove ambiguity

$A \text{-> } xP'$

$P' \text{ -> } P_1 \mid P_2 \mid P_3 \ldots \mid P_n$

- Example :

stmt -> *if* exp *then* stmt *endif* |
if exp *then* stmt *endif* *else* stmt *endif*

stmt -> *if* exp *then* stmt *endif* ELSEFUNC
ELSEFUNC -> *else* stmt *endif* | ϵ

Left Recursion

- A production is **left recursive** if its LHS symbol is the first symbol of its RHS.

- StmtList$\longrightarrow$ StmtList ; Stmt

  StmtList is left-recursion.
- Consider the left-recursive grammar $A \longrightarrow A\ \alpha \mid \beta$

A generates all strings starting with a β and followed by a number of α

- Can rewrite using right-recursion

  $A \longrightarrow \beta\ A'$

  $A' \longrightarrow \alpha\ A' \mid \varepsilon$

- In general,

$A \longrightarrow A\,\alpha_1 \mid \dots \mid A\,\alpha_n \mid \beta_1 \mid \dots \mid \beta_m$

Can be rewritten as

$A \longrightarrow \beta_1\, A' \mid \dots \mid \beta_m\, A'$

$A' \longrightarrow \alpha_1\, A' \mid \dots \mid \alpha_n\, A' \mid \varepsilon$

- Indirect left recursion

$S \longrightarrow A\,\alpha \mid \delta$

$A \longrightarrow S\,\beta$

As

$S \longrightarrow^{+} S\,\beta\,\alpha \mid \delta$

- Example

E⟶ E+T | T

T⟶ T*F | F

F (E) | id

After eliminating left recursion

E ⟶ T E'   ; E'--> +T E' | Є

T⟶ F T'   ; T'--> *FT' | ϵ

F ⟶ (E) | id

Parsers constructed after elimination of left recursion are recursive decent parsers

- easy to build,
- but inefficient, and might require backtracking

```
procedure E ( ) {
    T( );
    EPRIME( );
}
procedure T ( ) {
    F( );
    TPRIME( );
}
procedure EPRIME( ){
    if input = "+"
    {
        Advance( );
        T( );
        EPRIME( );
        return true;
    }
    return false
}
```

Recursive descent Parser

```
procedure TPRIME( ){
    if input = "*"
    {
        Advance( );
        F( );
        TPRIME( );
        return true;
    }
    return false
}
procedure F() {
    if input = "("
    {
        Advance( );
        E( );
        if input = ")"
                return true;
        else
                return false;
    }
    else if input = "id"
    {       Advance();
            return true;
    }
    return false
}
```

Home work

- **S $\rightarrow$ a | - | (T) | T**
- **T $\rightarrow$ T,S | a | - | (T)**