

Recursive Descent Parsing

Recursive Descent Parsing (RDP)

Reminder:

- Parser checks that its input (sequence of tokens) is syntactically correct
 - * not just a sequence of legal words
 - * sentence with correct structure: can be derived in the grammar of the language (derived from the initial variable S)

RDP:

- Tries to rebuild the syntax tree of the input
- The tree is rebuilt in a top-down way (from S downwards to tokens)
- Structure of parser reflects the structure of the language grammar
- If the grammar is recursive then the parser is recursive as well

How parser acts

Parser scans the stream of tokens from left to right:

t1 t2 t3 tn EOF



At every stage:

- **based on the history** (already seen in the input) :
updates its prediction (תחזית) for the rest –
what should come next for the input to be correct?
- checks: does the **reality** (tokens that it sees in the input) **fits the current prediction?**

Two types of predictions - 1

The next token is of kind t :

t1 t2 t3 tk t tn EOF
 \wedge

Checked by calling `match(t):`

```
void match(token_kind t)
{
    cur_token = next_token();
    if (cur_token -> kind != t)
        error();
}
```

This function **does not depend on the language's grammar**

Two types of predictions - 2

A fragment starting at the next token is derived from variable X :

t1 t2 t3 tk tk+1 ... tm.... tn EOF

^ |_____|
X

NOTE: input contains tokens only, so variable X itself doesn't appear in the input !

Checked by calling **parse_X()**

This function **does depend on the grammar**:

reflects the derivation rules for variable X ("תפזר" עבור X)

Initial prediction

Two requirements:

- the entire input is correctly derived from S
- after the derivation is finished, EOF is reached

Hence, parser's main function is as follows:

```
void parser()  
    {parse_S();  
      match(EOF)  
    }
```

Structure of parse_X()

1) Single derivation rule for variable

DECLARATION \rightarrow TYPE VAR_LIST

t1 t2 t3 tk tk+1 ... tm.... tn EOF

|_____|
DECLARATION

call parse_DECLARATION()

t1 t2 t3 tk tk+1 tm.... tn EOF

|TYPE VAR LIST|
DECLARATION

Function:

```
void parse_DECLARATION() { parse_TYPE(); parse_VAR_LIST() }
```

Structure of parse_X()

2) Several rules for same variable

$S \rightarrow id = E \mid \underline{\text{while } id < E \text{ do } S \text{ od}}$

$E \rightarrow id \mid \text{num}$

t1 t2 t3 tk tk+1 ... tm tn EOF

| _____ |
S

call parse_S()

can be

t1 t2 t3 tk tk+1 tm tn EOF

| id = E |
S

or

t1 t2 t3 tk tk+1 tm tn EOF

| while id < E do S od |
S

Structure of parse_X()

Function

```
parse_S() {  
    t = next_token();  
    switch(t) {  
        case id:  match(=);  
                  Parse_E();  
                  break;  
        case while: match(id);  
                  match(<);  
                  parse_E();  
                  match(do);  
                  parse_S();  
                  match(od);  
                  break;  
        default:  error();  
    }  
}
```

Explanation

- Call to parse_S(): prediction of a fragment derived from S
- case id: the next token is of type id
- Check: **can derivation from S start with id ?**
id \in First(S) ?
- Possible, if the rule $S \rightarrow id = E$ is used
- Update the prediction: after id, token = is expected, and then a fragment derived from E.

First(X)

First(X) = {t1, t2, ... , tn}

Group of tokens such that for every i ($1 \leq i \leq n$), there exists a derivation from X that starts with t_i .

In other words: derivation from X may start with t_i

Example: First(S) = {id , while}

Challenges

- Given G , how to compute First(X) for every variable X in G ?
(simple if every rule for X starts with a token)
- What if a rule starts with a variable, e.g. $X \rightarrow Y t_1 t_2 Z$?

Dealing with ϵ -rules

Example

1. $S \rightarrow a S b$
2. $S \rightarrow \#$
3. $S \rightarrow \epsilon$

- when parser applies the rule $S \rightarrow \epsilon$?
- ϵ is not a token ! (hence $\epsilon \notin \text{First}(S)$)

NOTE: also $b, \text{EOF} \notin \text{First}(S)$

```
void parse_S()
{
    t = next_token();
    switch(t -> kind) {
        case a: { print ("Rule 1"); parse_S();
                  match(b); break }
        case #: { print ("Rule 2"); break; }
        case b: error() ???
        case EOF: error() ???
    }
}
```

Dealing with ε -rules

Consider:

t1 t2 t3 tk b ... tm.... tn EOF

$$\Lambda \quad | \quad \text{---} \quad |$$

S

Is this an error? Reminder: $b \notin \text{First}(S)$

Can apply here the rule $S \rightarrow \varepsilon$!

t1 t2 t3 tk b ... tm.... tn EOF

$$\begin{array}{c} \wedge \\ | \\ \exists \end{array}$$

In fact, after t_k there is something (ε) correctly derived from S – as predicted!

```
void parse_S()
    t = next_token();
    switch(t -> kind) {
    case a: { print ("Rule 1"); parse_S();
             match(b); break }
    case #: { print ("Rule 2"); break; }
    case b: error() ???
    case EOF: error() ???
    }
}
```

Dealing with ϵ -rules

In fact, after t_k there is something correctly derived from S

$t_1 \ t_2 \ t_3 \ \dots \ t_k \ b \ \dots \ t_m \ \dots \ t_n \ \text{EOF}$

$\wedge \mid$
 ϵ

This is correct only if token b can appear

after derivation from S is completed

See rule 1 - this indeed is possible $b \in \text{Follow}(S)$

1. $S \rightarrow a S b$

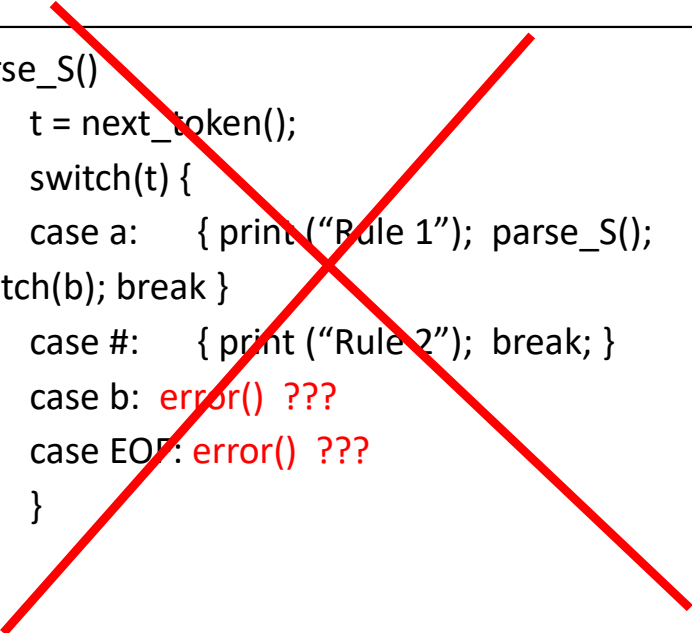
2. $S \rightarrow \#$

3. $S \rightarrow \epsilon$

Similar – for EOF

Dealing with ϵ -rules

```
void parse_S()
{
    t = next_token();
    switch(t) {
        case a: { print ("Rule 1"); parse_S();
        match(b); break }
        case #: { print ("Rule 2"); break; }
        case b: error() ???
        case EOF: error() ???
    }
}
```



```
void parse_S()
{
    t = next_token();
    switch(t -> kind) {
        case a: { print ("Rule 1"); parse_S();
        match(b); break }
        case #: { print ("Rule 2"); break; }
        case b: case EOF:
            { print ("Rule 3");
            t = back_token(); break; }
    }
}
```

/ NOTE: cases also for tokens in Follow(S) ! */*

Why back_token?

- next_token returned token b
- b is not a part of what is derived from S (derived ϵ)
- parser made a step **beyond** S
- since parser is still inside parse_S (means: inside S), it should step back

Follow(X)

Follow(X) = {t1, t2, ... , tn}

Group of tokens such that for every i ($1 \leq i \leq n$), there exists a derivation in which t_i appears after X

Example: Follow(S) = {b , EOF}

Challenges

- Given G, how to compute Follow(X) for every variable X in G ?
(simple if every appearance of X is immediately followed by a token)
- What to do in cases such as $Y \rightarrow t_1 X Z$, or $Y \rightarrow t_1 X$?

Can recursive descent parser be constructed for every grammar?

No, there are some obstacles to be eliminated:

- Left common prefixes
- Left recursion

Left common prefixes – why this is a problem?

Example

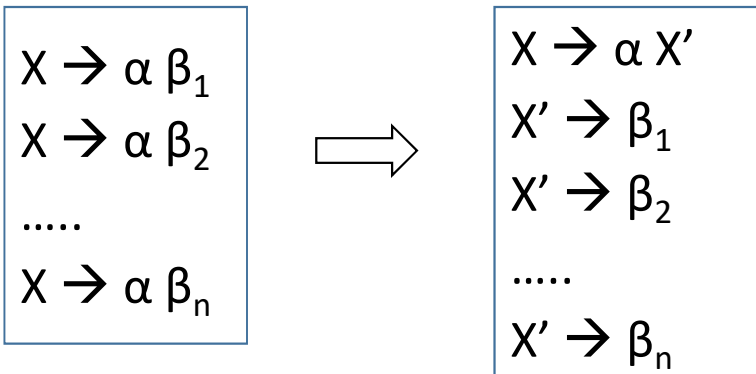
DECL \rightarrow TYPE LIST
TYPE \rightarrow int | real
LIST \rightarrow id | id , LIST

Left common prefix in rules for LIST:

two rules for same variable have a common beginning

```
parse_LIST()
{
    t = next_token();
    switch (t -> kind) {
        case id : ???? /* id  $\in$  First(LIST) , but... not clear which rule for LIST to apply */
        default : error()
    }
}
```

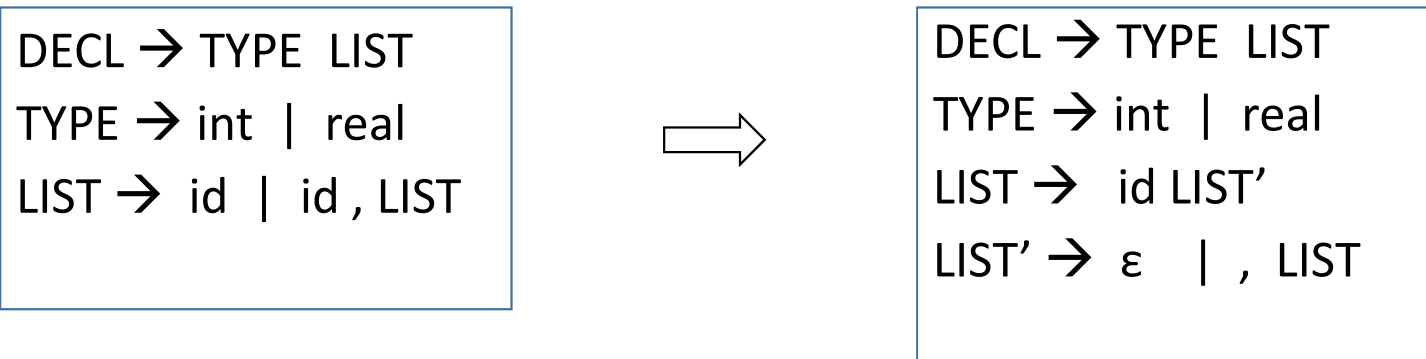
Elimination of left common prefixes



Need:

- add a new variable X'
- replace rules by new ones
(an ϵ -rule may appear !)

Example



Left recursion – why this is a problem?

Example

DECL \rightarrow TYPE LIST
TYPE \rightarrow int | real
LIST \rightarrow id | LIST, id

Left recursion on LIST

```
parse_LIST()
{
    t = next_token();
    switch (t -> kind) {
        case id : ??? not clear which rule for LIST to apply
        default : error()
    }
}
```

Elimination of left recursion

$$\begin{array}{l} X \rightarrow X\alpha \\ X \rightarrow \beta \end{array}$$

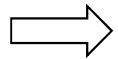
Generates sequences of the form $\beta\alpha^n$ ($n \geq 0$)

First, α 's are produced; at the end of derivation β is added

Examples: $X \rightarrow \beta$ $X \rightarrow X\alpha \rightarrow \beta\alpha$ $X \rightarrow X\alpha \rightarrow X\alpha\alpha \rightarrow \beta\alpha\alpha$

Idea: first produce β , and then continue to production of α 's

$$\begin{array}{l} X \rightarrow X\alpha \\ X \rightarrow \beta \end{array}$$



$$\begin{array}{l} X \rightarrow \beta X' \\ X' \rightarrow \epsilon \\ X' \rightarrow \alpha X' \end{array}$$

Need:

- add a new variable X'
- replace rules by new ones (ϵ -rule must appear!)

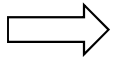
$$X \rightarrow \beta X' \rightarrow (X' \rightarrow \epsilon) \rightarrow \beta$$

$$X \rightarrow \beta X' \rightarrow (X' \rightarrow \alpha X') \rightarrow \beta\alpha X' \rightarrow (X' \rightarrow \epsilon) \rightarrow \beta\alpha$$

$$X \rightarrow \beta X' \rightarrow (X' \rightarrow \alpha X') \rightarrow \beta\alpha X' \rightarrow (X' \rightarrow \alpha X') \rightarrow \beta\alpha\alpha X' \rightarrow (X' \rightarrow \epsilon) \rightarrow \beta\alpha\alpha$$

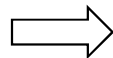
Elimination of left recursion - example

$X \rightarrow X\alpha$
 $X \rightarrow \beta$



$X \rightarrow \beta X'$
 $X' \rightarrow \epsilon$
 $X' \rightarrow \alpha X'$

$DECL \rightarrow TYPE\ LIST$
 $TYPE \rightarrow int \mid real$
 $LIST \rightarrow id \mid LIST, id$



$DECL \rightarrow TYPE\ LIST$
 $TYPE \rightarrow int \mid real$
 $LIST \rightarrow id\ LIST'$
 $LIST' \rightarrow \epsilon \mid LIST' \rightarrow , id\ LIST'$

Left recursion on LIST : $\alpha = , id$; $\beta = id$