

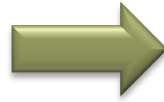
Grammar vs Recursive Descent Parser

```
expr ::= term termList
termList ::= + term termList
           | - term termList
           | ε
term ::= factor factorList
factorList ::= * factor factorList
            | / factor factorList
            | ε
factor ::= name | ( expr )
name ::= ident
```

```
def expr = { term; termList }
def termList =
  if (token==PLUS) {
    skip(PLUS); term; termList
  } else if (token==MINUS)
    skip(MINUS); term; termList
  }
def term = { factor; factorList }
...
def factor =
  if (token==IDENT) name
  else if (token==OPAR) {
    skip(OPAR); expr; skip(CPAR)
  } else error("expected ident or ")
```

Rough General Idea

$A ::= B_1 \dots B_p$
| $C_1 \dots C_q$
| $D_1 \dots D_r$



```
def A =  
  if (token ∈ T1) {  
    B1 ... Bp  
  } else if (token ∈ T2) {  
    C1 ... Cq  
  } else if (token ∈ T3) {  
    D1 ... Dr  
  } else error("expected T1,T2,T3")
```

where:

$T1 = \mathbf{first}(B_1 \dots B_p)$

$T2 = \mathbf{first}(C_1 \dots C_q)$

$T3 = \mathbf{first}(D_1 \dots D_r)$

$\mathbf{first}(B_1 \dots B_p) = \{a \in \Sigma \mid B_1 \dots B_p \Rightarrow \dots \Rightarrow aw\}$

$T1, T2, T3$ should be **disjoint** sets of tokens.

Computing **first** in the example

```
expr ::= term termList
termList ::= + term termList
           | - term termList
           | ε
term ::= factor factorList
factorList ::= * factor factorList
            | / factor factorList
            | ε
factor ::= name | ( expr )
name ::= ident
```

$\text{first}(\text{name}) = \{\mathbf{ident}\}$

$\text{first}(\text{(expr)}) = \{ (\ }$

$\text{first}(\text{factor}) = \text{first}(\text{name})$

$\cup \text{first}(\text{(expr)})$

$= \{\mathbf{ident}\} \cup \{ (\ }$

$= \{\mathbf{ident}, (\ }$

$\text{first}(* \text{ factor factorList}) = \{ * \}$

$\text{first}(/ \text{ factor factorList}) = \{ / \}$

$\text{first}(\text{factorList}) = \{ *, / \}$

$\text{first}(\text{term}) = \text{first}(\text{factor}) = \{\mathbf{ident}, (\ }$

$\text{first}(\text{termList}) = \{ +, - \}$

$\text{first}(\text{expr}) = \text{first}(\text{term}) = \{\mathbf{ident}, (\ }$

Algorithm for **first**

Given an arbitrary context-free grammar with a set of rules of the form $X ::= Y_1 \dots Y_n$ compute first for each right-hand side and for each symbol.

How to handle

- alternatives for one non-terminal
- sequences of symbols
- nullable non-terminals
- recursion

Rules with Multiple Alternatives

$$A ::= B_1 \dots B_p \\ | C_1 \dots C_q \\ | D_1 \dots D_r$$

$$\text{first}(A) = \text{first}(B_1 \dots B_p) \\ \cup \text{first}(C_1 \dots C_q) \\ \cup \text{first}(D_1 \dots D_r)$$

Sequences

$$\text{first}(B_1 \dots B_p) = \text{first}(B_1)$$

if not nullable(B_1)

$$\text{first}(B_1 \dots B_p) = \text{first}(B_1) \cup \dots \cup \text{first}(B_k)$$

if nullable(B_1), ..., nullable(B_{k-1}) and
not nullable(B_k) or $k=p$

Abstracting into Constraints

recursive grammar: constraints over finite sets: expr' is $\text{first}(\text{expr})$

```
expr ::= term termList
termList ::= + term termList
           | - term termList
           | ε
term ::= factor factorList
factorList ::= * factor factorList
            | / factor factorList
            | ε
factor ::= name | ( expr )
name ::= ident
```

nullable: termList, factorList

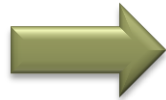
```
expr' = term'
termList' = {+}
           U {-}

term' = factor'
factorList' = {*}
            U {/}

factor' = name' U { ( }
name' = { ident }
```

For this nice grammar, there is no recursion in constraints. Solve by substitution.

Example to Generate Constraints

$$\begin{aligned} S &::= X \mid Y \\ X &::= \mathbf{b} \mid S Y \\ Y &::= Z X \mathbf{b} \mid Y \mathbf{b} \\ Z &::= \varepsilon \mid \mathbf{a} \end{aligned}$$

$$\begin{aligned} S' &= X' \cup Y' \\ X' &= \end{aligned}$$

terminals: \mathbf{a}, \mathbf{b}

non-terminals: S, X, Y, Z

reachable (from S):

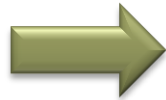
productive:

nullable:

First sets of terminals:

$$S', X', Y', Z' \subseteq \{\mathbf{a}, \mathbf{b}\}$$

Example to Generate Constraints

$$\begin{aligned} S &::= X \mid Y \\ X &::= \mathbf{b} \mid S Y \\ Y &::= Z X \mathbf{b} \mid Y \mathbf{b} \\ Z &::= \varepsilon \mid \mathbf{a} \end{aligned}$$

$$\begin{aligned} S' &= X' \cup Y' \\ X' &= \{\mathbf{b}\} \cup S' \\ Y' &= Z' \cup X' \cup Y' \\ Z' &= \{\mathbf{a}\} \end{aligned}$$

terminals: \mathbf{a}, \mathbf{b}

non-terminals: S, X, Y, Z

reachable (from S): S, X, Y, Z

productive: X, Z, S, Y

nullable: Z

These constraints are recursive.
How to solve them?

$$S', X', Y', Z' \subseteq \{\mathbf{a}, \mathbf{b}\}$$

How many candidate solutions

- in this case?
- for k tokens, n nonterminals?

Iterative Solution of **first** Constraints

	S'	X'	Y'	Z'
1.	$\{\}$	$\{\}$	$\{\}$	$\{\}$
2.	$\{\}$	$\{b\}$	$\{b\}$	$\{a\}$
3.	$\{b\}$	$\{b\}$	$\{a,b\}$	$\{a\}$
4.	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a\}$
5.	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a\}$

$$\begin{aligned} S' &= X' \cup Y' \\ X' &= \{b\} \cup S' \\ Y' &= Z' \cup X' \cup Y' \\ Z' &= \{a\} \end{aligned}$$

- Start from all sets empty.
- Evaluate right-hand side and assign it to left-hand side.
- Repeat until it stabilizes.

Sets grow in each step

- initially they are empty, so they can only grow
- if sets grow, the RHS grows (U is monotonic), and so does LHS
- they cannot grow forever: in the worst case contain all tokens

Constraints for Computing Nullable

- Non-terminal is nullable if it can derive ε

$$\begin{aligned} S &::= X \mid Y \\ X &::= \mathbf{b} \mid S Y \\ Y &::= Z X \mathbf{b} \mid Y \mathbf{b} \\ Z &::= \varepsilon \mid \mathbf{a} \end{aligned}$$

$$\begin{aligned} S' &= X' \mid Y' \\ X' &= 0 \mid (S' \& Y') \\ Y' &= (Z' \& X' \& 0) \mid (Y' \& 0) \\ Z' &= 1 \mid 0 \end{aligned}$$

$S', X', Y', Z' \in \{0,1\}$

0 - not nullable

1 - nullable

| - disjunction

& - conjunction

	S'	X'	Y'	Z'
1.	0	0	0	0
2.	0	0	0	1
3.	0	0	0	1

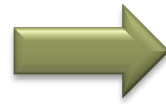
again monotonically growing

Computing first and nullable

- Given any grammar we can compute
 - for each non-terminal X whether $\text{nullable}(X)$
 - using this, the set $\text{first}(X)$ for each non-terminal X
- General approach:
 - generate constraints over finite domains, following the structure of each rule
 - solve the constraints iteratively
 - start from least elements
 - keep evaluating RHS and re-assigning the value to LHS
 - stop when there is no more change

Rough General Idea

```
A ::= B1 ... Bp
      | C1 ... Cq
      | D1 ... Dr
```



```
def A =
  if (token ∈ T1) {
    B1 ... Bp
  } else if (token ∈ T2) {
    C1 ... Cq
  } else if (token ∈ T3) {
    D1 ... Dr
  } else error("expected T1,T2,T3")
```

where:

T1 = **first**(B₁ ... B_p)

T2 = **first**(C₁ ... C_q)

T3 = **first**(D₁ ... D_r)

T1, T2, T3 should be **disjoint** sets of tokens.

Exercise 1

$A ::= B \text{ EOF}$

$B ::= \varepsilon \mid B B \mid (B)$

A	B	← nullable
0	0	
0	1	

- Tokens: **EOF**, (,)
- Generate constraints and compute nullable and first for this grammar.
- Check whether first sets for different alternatives are disjoint.

Exercise 2

$S ::= B \text{ EOF}$

$B ::= \varepsilon \mid B (B)$

- Tokens: **EOF**, (,)
- Generate constraints and compute nullable and first for this grammar.
- Check whether first sets for different alternatives are disjoint.

Exercise 3

Compute nullable, first for this grammar:

$\text{stmtList} ::= \varepsilon \mid \text{stmt stmtList}$

$\text{stmt} ::= \text{assign} \mid \text{block}$

$\text{assign} ::= \text{ID} = \text{ID} ;$

$\text{block} ::= \text{beginof ID stmtList ID ends}$

Describe a parser for this grammar and explain how it behaves on this input:

beginof myPrettyCode

x = u;

y = v;

myPrettyCode **ends**

Problem Identified

$\text{stmtList} ::= \varepsilon \mid \text{stmt stmtList}$

$\text{stmt} ::= \text{assign} \mid \text{block}$

$\text{assign} ::= \mathbf{ID} = \mathbf{ID} ;$

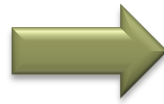
$\text{block} ::= \mathbf{beginof ID stmtList ID ends}$

Problem parsing stmtList :

- \mathbf{ID} could start alternative stmt stmtList
- \mathbf{ID} could **follow** stmt , so we may wish to parse ε that is, do nothing and return
- For nullable non-terminals, we must also compute what follows them

General Idea for nullable(A)

```
A ::= B1 ... Bp
      | C1 ... Cq
      | D1 ... Dr
```



```
def A =
  if (token ∈ T1) {
    B1 ... Bp
  } else if (token ∈ (T2 ∪ TF)) {
    C1 ... Cq
  } else if (token ∈ T3) {
    D1 ... Dr
  } // no else error, just return
```

where:

T1 = **first**(B₁ ... B_p)

T2 = **first**(C₁ ... C_q)

T3 = **first**(D₁ ... D_r)

T_F = **follow**(A)

Only one of the alternatives can be nullable (e.g. second)
T1, T2, T3, T_F should be pairwise **disjoint** sets of tokens.

LL(1) Grammar - good for building recursive descent parsers

- Grammar is LL(1) if for each nonterminal X
 - first sets of different alternatives of X are disjoint
 - if nullable(X), first(X) must be disjoint from follow(X)
- For each LL(1) grammar we can build recursive-descent parser
- Each LL(1) grammar is unambiguous
- If a grammar is not LL(1), we can sometimes transform it into equivalent LL(1) grammar

Computing if a token can follow

first($B_1 \dots B_p$) = $\{a \in \Sigma \mid B_1 \dots B_p \Rightarrow \dots \Rightarrow aw\}$

follow(X) = $\{a \in \Sigma \mid S \Rightarrow \dots \Rightarrow \dots Xa \dots\}$

There exists a derivation from the start symbol that produces a sequence of terminals and nonterminals of the form $\dots Xa \dots$
(the token a follows the non-terminal X)

Rule for Computing Follow

Given $X ::= YZ$ (for reachable X)

then $\mathbf{first}(Z) \subseteq \mathbf{follow}(Y)$

and $\mathbf{follow}(X) \subseteq \mathbf{follow}(Z)$

now take care of nullable ones as well:

For each rule $X ::= Y_1 \dots Y_p \dots Y_q \dots Y_r$

$\mathbf{follow}(Y_p)$ should contain:

- $\mathbf{first}(Y_{p+1}Y_{p+2}\dots Y_r)$
- also $\mathbf{follow}(X)$ if $\mathbf{nullable}(Y_{p+1}Y_{p+2}Y_r)$

Compute nullable, first, follow

stmtList ::= ε | stmt stmtList

stmt ::= assign | block

assign ::= **ID = ID ;**

block ::= **beginof ID stmtList ID ends**

Is this grammar LL(1)?

Conclusion of the Solution

The grammar is not LL(1) because we have

- nullable(stmtList)
- $\text{first}(\text{stmt}) \cap \text{follow}(\text{stmtList}) = \{\mathbf{ID}\}$
- If a recursive-descent parser sees **ID**, it does not know if it should
 - finish parsing stmtList or
 - parse another stmt