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

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

→

def A =
  if (token ∈ T1) {
    $B_1 \ldots B_p$
  else if (token ∈ T2) {
    $C_1 \ldots C_q$
  } **else if** (token ∈ T3) {
    $D_1 \ldots D_r$
  } **else** error("expected T1,T2,T3")

where:

T1 = **first**($B_1 \ldots B_p$)
T2 = **first**($C_1 \ldots C_q$)
T3 = **first**($D_1 \ldots D_r$)

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

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**

first(name) = {**ident**}
first(**(** expr **)**) = { **(** }
first(factor) = first(name)
U first(**(** expr **)**)
= {**ident**} U{ **(** }
= {**ident**, **(** }
first(***** factor factorList) = { ***** }
first(**/** factor factorList) = { **/** }
first(factorList) = { *****, **/** }
first(term) = first(factor) = {**ident**, **(** }
first(termList) = { **+** , **-** }
first(expr) = first(term) = {**ident**, **(** }

Algorithm for **first**

Given an arbitrary context-free grammar with a set of rules of the form X ::= $Y_1 \ldots Y_n$ 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

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

→

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

Sequences

$$first(B_1 \dots B_p) = first(B_1)$$

if not nullable(B_1)

$$first(B_1 \dots B_p) = first(B_1) \cup \dots \cup first(B_k)$$

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 first(expr)

recursive grammar	constraints
expr ::= term termList	expr' = term'
termList ::= **+** term termList	termList' = {**+**}
| **-** term termList	U {**-**}
| ε	
term ::= factor factorList	term' = factor'
factorList ::= ***** factor factorList	factorList' = {*****}
| **/** factor factorList	U { **/** }
| ε	
factor ::= name | **(** expr **)**	factor' = name' U { **(** }
name ::= **ident**	name' = { **ident** }

nullable: termList, factorList

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

Example to Generate Constraints

S ::= X | Y
X ::= **b** | S Y
Y ::= Z X **b** | Y **b**
Z ::= ε | **a**

S' = X' U Y'
X' =

terminals: **a**,**b**
non-terminals: S, X, Y, Z

reachable (from S):
productive:
nullable:

First sets of terminals:
S', X', Y', Z' $\subseteq$ {a,b}

Example to Generate Constraints

S ::= X | Y
X ::= **b** | S Y
Y ::= Z X **b** | Y **b**
Z ::= ε | **a**

$\Rightarrow$

S' = X' U Y'
X' = {b} U S'
Y' = Z' U **X'** U Y'
Z' = {a}

terminals: **a**,**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?

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

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}

S' = X' U Y'
X' = {b} U S'
Y' = Z' U **X'** U Y'
Z' = {a}

- 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

Grammar		Constraints
S ::= X | Y	⇒	S' = X' | Y'
X ::= **b** | S Y		X' = 0 | (S' & Y')
Y ::= Z X **b** | Y **b**		Y' = (Z' & X' & 0) | (Y' & 0)
Z ::= ε | **a**		Z' = 1 | 0

S', X', Y', Z' ∈ {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 nullable(X)
  - using this, the set 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

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

def A =
  if (token ∈ T1) {
    $B_1 \ldots B_p$
  else if (token ∈ T2) {
    $C_1 \ldots C_q$
  } **else if** (token ∈ T3) {
    $D_1 \ldots D_r$
  } **else** error("expected T1,T2,T3")

where:

T1 = **first**($B_1 \ldots B_p$)
T2 = **first**($C_1 \ldots C_q$)
T3 = **first**($D_1 \ldots D_r$)

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

Exercise 1

A ::= B **EOF**

B ::= ε | B B | **(**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 **EOF**

B ::= ε | 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:

stmtList ::= ε | stmt stmtList

stmt ::= assign | block

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

block ::= **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

stmtList ::= ε | stmt stmtList

stmt ::= assign | block

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

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

Problem parsing stmtList:

- **ID** could start alternative stmt stmtList
- **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)

$$A ::= B_1 \ldots B_p \mid C_1 \ldots C_q \mid D_1 \ldots D_r$$

→

def A =
  if (token ∈ T1) {
    $B_1 \ldots B_p$
  else if (token ∈ **(**T2 U T_F)) {
    $C_1 \ldots C_q$
  } **else if** (token ∈ T3) {
    $D_1 \ldots D_r$
  } // no else error, just return

where:

T1 = **first**($B_1 \ldots B_p$)
T2 = **first**($C_1 \ldots C_q$)
T3 = **first**($D_1 \ldots D_r$)
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$(B_1 \ldots B_p) = \{a \in \Sigma \mid B_1 \ldots B_p \Rightarrow \ldots \Rightarrow aw\}$

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

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

Rule for Computing Follow

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

then **first**(Z) $\subseteq$ **follow**(Y)

and **follow**(X) $\subseteq$ **follow**(Z)

now take care of nullable ones as well:

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

follow(Y_p) should contain:

- **first(**$Y_{p+1} Y_{p+2} \ldots Y_r$)
- also **follow**(X) if 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)
- first(stmt) ∩ follow(stmtList) = {**ID**}

- If a recursive-descent parser sees **ID**, it does not know if it should
  - finish parsing stmtList or
  - parse another stmt